MONOGRAPHS IN COMPUTER SCIENCE

PARSING TECHNIQUES 解析技术实用指南（中文版）

A Practical Guide

Dick Grune
Ceriel J.H. Jacobs

Second Edition

Springer

目錄

解析技术

本书介绍

《解析技术》是编译器前端经典书籍《Parsing Techniques》的中文译本。

作者

- Dick Grune
- Ceriel J.H. Jacobs

译者

- Agnes Huang
- Rex Lee

Github仓库

https://github.com/duguying/parsing-techniques

翻译进度

18% [125/677]

加入协作

- 加QQ群 242838077
- 认领章节
- fork仓库

- PR

捐助

支付宝



微信



请各位捐助时备注一下称呼。以便能在列表中记录。感谢！

捐献列表

昵称	金额	日期
独孤影	￥10.00	2016/10/10
luoyinjiexx	￥10.00	2016/10/10
Hanruis	￥10.00	2017/05/10
yetone	￥10.00	2017/06/22
yetone	￥10.00	2017/06/23
Laily	￥10.00	2017/07/06
飞龙	￥20.00	2017/07/17
关大龙	￥1.68	2017/07/17
liman1008	￥2.00	2017/07/18
春晓	￥10.00	2017/08/07
匿名	￥50.00	2017/08/12
匿名	￥10.00	2017/11/18
victor_xie	￥20.00	2017/12/28
匿名	￥20.00	2018/01/13
奕普	￥10.00	2018/03/04
匿名	￥10.00	2018/03/15
匿名	￥10.00	2018/04/10
jello	￥20.00	2018/05/07
Suzu	￥10.00	2018/05/29

第二版前言

为了满足读者的需求，同时也是我们自身的愿望，我们出版了第二版。尽管解析技术不是快速发展的领域，但是他依然在向前发展。当第一版在1990年出版的时候，只有一个简单的并且局限性相当大的线性时间字符串解析算法(algorithm for linear-time substring parsing)。但是发展到现在已经有多种可以覆盖所有确定语言的强大的算法；我们将在第12章对这些算法进行详细介绍。在1990年，Theorem 8.1诞生于一篇1961年由Bar-Hillel, Perles, 和Shamir发表的至今已经落满了灰尘的论文；在过去的十年里，Theorem 8.1被用于创建新的算法，并且使现有的算法得到加强。我们在第13章中提及此事。

越来越多的非Chomsky(non-Chomsky)文法的系统被使用，语言学上尤为明显。20年前，除了两级语法之外没有任何建树，而现在却不再是那样，我们在第15章描述了其中6个非Chomsky文法的系统。曾经，非标准解析器被认为是很古怪的；如今，它们是我们所拥有的最强大的线性时间解析器(linear-time parsers)之一，在第10章我们再来说这个。

虽然还不是很实用，但是以另一种方式展现解析原理的奇妙并行解析算法，已经出现在人们的视野中，我们将在第14章详细介绍。在1990年还被认为不可能出现的广义LL语法分析器，现在也已经成为了现实，我们将在第11章介绍其中的2个。

一直以来，解析器都被用于解析；现如今，解析器已经被用于代码生成、数据压缩以及逻辑语言实现等方面，如第17.5节所示。而且，读者还能够在书目的第18章找到更多的开发案例。

Kees van Reeuwijk曾经半开玩笑地称呼我们的书为“濒危语法分析算法最后的栖息地”。我们同意这种说法，但是不完全同意，因为它远不止于此，对于这个说法我们受之无愧。在这本书中收录的的一些算法，它们有非常大的局限性甚至根本没有实用价值。不过我们依旧收录了它们，因为我们

觉得这些算法的思路很有意思，并且可以给我们以启迪，并且这些算法还有成长和发展的空间。我们也同样收录了许多有实用价值但是很少被用到的算法，希望将这些算法收录到书中能改变一下它们的现状。

关于练习和问题

这本书不是学校中所谓的教科书。少数大学开设了解析技术的课程，并且，如第一版序言所述，本书并不是只为某一类型的读者所准备。因此，书中几乎没有布置任何习题或作业，读者们可以自己为自己设置一些。章节末尾的问题，只是为了让读者不要陷入书中世界。我们将章节末的问题大致划分为三个类型：

- 无标记型 —— 短时间内可以完成的项目。
- 普通标记型 —— 需要花费一些时间和一些精力来完成的项目。
- 标记未研究型 —— 需要耗费大量时间精力，但目前我们尚未做到的项目。不过我们希望有读者能完成。

对于这些课题我们没有特意使之存在联系性，我们只是希望读者能从中得到启发、乐趣、或任何有用的东西。关于问题的思路、提示、部分或者完整的解决方案，见A章节。

关于正式语言（formal language）也有几个问题，不过我们没有在现有的文献中得到解答，可是却对解析有一定的重要性。这些问题已经在章节末标识出来了。

关于参考文献

第一版中，我们作为作者，阅读分析了所有我们能获得得资料文献。17年后，随着各类出版物得增加，以及互联网带来得查阅便利，查阅分析资料终于不再是一件让我们头疼得事情。在第一版的参考文献中，我们列举了全部的资料。但是在第二版中，尽数列举却不在可能，因为这些文件太多以至于全部列举将会占用远超本书主体内容的空间。在印刷版中，我们只列举了我们引用了原文的资料。我们把完整的参考资料放在了本书的网站

中，并且还有我们写的摘要和主题索引。因为建立了网站，我们才能没有顾虑的将全部的文章列举出来，并且可以列举一些平常不太容易获取的资料。网站上的章节是第18章，我们叫做“电子章节”。

本书没有提供URL，有两个原因：首先URL并不稳定，也许过一段时间，这些URL就不在指向我们的资料了；还有，特别是对软件来说，当你在阅读本书的时候也许就有了更好的下载地址。其实我们认为，提供URL还不如提供一些搜索关键字更好，关键字能帮你在搜索引擎上找到你想要的信息。

最近10年里，我们能看到有很多的博士论文使用的不是英语，以德语、法语、西班牙语还有爱沙尼亚语为主。然后很遗憾，这些论文由于语言的选择而难以被科学界所发现。这对相关科学领域以及对作者本人来说，都是一大损失。忽略个人情感，我们得承认，英语是现在科学界的主流语言。在过去的150年时间里，对科学感兴趣的绅士们，在休闲之余会读一些法语、德语、英语、希腊语、拉丁语或者是一点点梵语的东西；但是现在，学生和科学家们则希望花费更多的精力来吸收海量的知识。即便我们能够读懂大部分的上述语言（并非全部）所写的内容，并尽力将这些论文的精华传达给读者，但这仍然不足以让这些论文拥有它们应得的位置。

关于解析技术的未来

如果将来有幸能出版第三版，我们将会致力于精简内容，尽量的精细化（除了参考文献部分）。因为解析算法都大同小异，当你进行深入研究后就会有这样的发现，并且似乎有大一统的趋势。大体来说，差不多所有的解析都是在左递归保护下通过自顶向下的搜索来完成的；这么说是没有问题的，即便是传统的自底向上搜索技术也是如此，比如LR（1），而且自顶向下搜索技术是建立在LR（1）解析表的基础之上的。Earley方法（Earley's method）很明显是通过一些方式来区分自顶向下搜索和自底向上搜索的。表格解析的记忆机制，可以有效的减轻指数级别的搜索所带来的疼痛。并且它似乎可以在将深度优先搜索算法转化为广度优先搜索算法过程中，产生许多广义确定性算法；这方面在Sikkel的博士论文[158]中有

详细论述。上述汇集起来基本上就涵盖了本书提到的所有算法，包括学科交叉处涉及的解析算法。存粹的自底向上，不包含自顶向下组件的解析器，是很罕见的并且功能也不强大。

关于解析理论的未来，可以预见算法的统一将会大大的简化解析的复杂程度；而在领域交叉处，解析技术将会扮演一个上面样的角色尚不得而知。这种简化似乎并不会延伸到正式语言领域：它将依旧难以使用直观的事实来证明所有LL（1）语法就是LR（1），就像35年前一样。

解析技术最实际的未来可能潜藏在先进的模式识别中，但不包含这个模式的默认任务；然而，在领域交叉处，解析技术所将扮演的角色依旧不甚明朗。

Dick Grune

Ceriel J.H. Jacobs

2007年6月 于Amsterdam, Amstelveen

致谢

感谢E. Bermudez, Stuart Broad, Peter Bumbulis, Salvador Cavadini, Carl Cerecke, Julia Dain, Akim Demaille, Matthew Estes,Wan Fokkink, Brian Ford, Richard Frost, Clemens Grabmayer, Robert Grimm, Karin Harbusch, Stephen Horne, Jaco Imthorn, Quinn Tyler Jackson, Adrian Johnstone, Michiel Koens, Jaroslav Král, Olivier Lecarme, Lillian Lee, Olivier Lefevre, Joop Leo, JianHua Li, Neil Mitchell, Peter Pepper, Wim Pijls, José F. Quesada, Kees van Reeuwijk, Walter L. Ruzzo, Lothar Schmitz, Sylvain Schmitz, Thomas Schoebel-Theuer, Klaas Sikkel, Michael Sperberg-McQueen, Michal Žemliˇcka, Hans Åberg，等等，感谢他们寄给我们关于第一版的信件，评论和勘误表。并且在此特别感谢阅读本书初稿的Kees van Reeuwijk 和Sylvain Schmitz当然还有我们这些作者，审阅初稿对本书的出版有着重大的作用。

感谢Faculteit Wiskunde en Informatica of the Vrije Universiteit 允许我们使用了他们设备。

同时，感谢本书结尾处列出的将近1500位作者以及他们所研究的美妙的算法，并且让我们可以使用这些算法。这本书的每一页内容，都是建立在这些算法的基础上的。

第一版前言

语法解析（句法分析）是计算机科学中最好理解分支机构之一。解析器已经被广泛应用于大量的学科：例如，计算机科学（编译器构造、数据库接口、自描述数据库、人工智能），语言学（文本分析、语料库分析、机器翻译、圣经的文本分析）、文档编制和格式转换、排版化学公式中和在染色体识别等等;解析器可以在大量的学科中使用（也许已经在使用中了）。因此，现今还没有一本向非专业人士系统介绍解析技术的书籍出版，真的是一件很奇怪的事情。其中的一个原因可能是，解析素来被认为是一个“难题”。不过，根据我们在研究阿姆斯特丹编译器（Amsterdam Compiler Kit）以及在讲解编译器构造的经验，解析技术可以通过一些简明的方式来解释说明。本书就是在这样的情况下出版的。

本书并不是只单一的面向某一部分读者。相反，在编写时，我们一直在努力让本书适合于学生、各学科的老师、各个行业不懂解析的人、各种科学报刊的狂热读者、等等等等。这项技术将很难以课程形式出现，因为对于这种定时开讲的内容固定的教学模式，并不适合上面提到的复杂的读者群体；这就是我们出版这本书的目的，它可以让你在需要的时候，在任何时间任何地点没有障碍的获取你需要的知识。

要应对这样一个复杂的读者群体，是有一定的困难的（当然也有好处）。本书虽然没有明显的使用数学理论，但不可避免的书中也蕴含了大量的额数学思想。大部分解析技术的专业术语在书中都给出了明确的定义，但仍然有部分处于学科边缘的术语我们没有进行定义。每一位参与过非自己研究领域的主题研究的读者，应该都能明白这种情况。读者们可以跳过这些术语，或者用自己相对熟悉的东西来进行替代，然后你当然不希望这个术语出现的太频繁。当然书中也会有一些地方，会让读者觉得讲述的非常明确易懂（也许本段就是这样的）。有一点可能会让读者们感到欣慰的就是，阅读这本书并不会是浪费时间，或者像你在上一堂专业课时只能盯着窗外那样难以理解老师讲的内容，而只能消磨一下时间。

我们写作这本书的主要目的，希望通过出版书籍的方式，来揭示隐藏于于某些文字表面下的解析的存在，就像这个公式：

> Let □ be a mapping □ and □ a homomorphism

读者并不需要一定懂得某种编程语言。书中确实有两三个Pascal语言的程序，不过只是用来展示而已，在对解析的阐述中这个并不重要。真正需要具有的时对算法的了解，尤其时递归算法（recursion）。Howard Johnston (Prentice-Hall, 1985)写的*Learning to program*或者Richard Bornat (Prentice-Hall 1987)写的*Programming from first principles*，这些书已经为我们提供了足够多的知识储备（虽然似乎有些过分的详细了）。选择Pascal，是因为这差不多是唯一一门在计算机领域之外被使用的编程语言了。

结尾处符的大量参考文献应该是本书的一大特色了。对参考文献感兴趣的读者可能比我们预期的还要多，尤其是对本书中所提及的某些领域有所了解的读者，不管是不是通过这本书所产生的了解。参考文献通过列表形式展示，这样更利于读者查找；批注放在了文章的页脚，并且也在书尾列出，目的是希望这些批注能像里程碑一样可以帮助读者在阅读时更好的理解。

关于应用程序的解析器，本书不做详细阐述。虽然我们在第一章提到了很多应用程序，但是由于我们没有相关的专业知识以提供更多的详细说明。虽然音乐作品所拥有的文法结构在很大程度上和解析相同，但是我们并不打算在这里对音乐进行探讨，这件事就留给音乐家们来做吧。同样的，虽然企业的营收与各种规章制度的文法似乎关系不大，但是我们坚信这之间也有存在某些联系的，不过这将是社会心理研究的课题。

致谢

在此感谢在我们编写本书的过程中，为我们提供过帮助的人们。Marion de Krieger 为我们检索了无数的书籍和期刊文章的复印版，正是有了她的努力，我们才能毫不费力的将书目收纳的如此完备。Ed Keizer 做好了我们和pic|tbl|eqn|psfig|troff这一堆乱七八糟的东西之间的修复工作，他无数次的

纠正了被我们滥用、重用或者明显被误用了的地方。Leo van Moergestel 使用硬件资源为我们完成了大量的工作，而这是我们这些门外汉所不会使用的。同时我们也要感谢Erik Baalbergen, Frans Kaashoek, Erik Groeneveld, Gerco Ballintijn , Jaco Imthorn 以及Egon Amada，他们为我们提供了大量的准确的批注。本书结尾的倒数第二章，出自Arwen Grune 。而Ilana和Lily Grune 为我们做了大量的录入工作。

感谢Faculteit Wiskunde en Informatica of the Vrije Universiteit 允许我们使用了他们设备。

同时，感谢数百位的专家，他们的作品为我们提供了那么多巧妙而精致的算法，并给出了这些算法的使用技巧。希望在我们列举的参考书目中没有遗漏任何一位。

Dick Grune

Ceriel J.H. Jacobs

1990年7月 于Amsterdam, Amstelveen

1 简介

解析是对一种给定语法构建其线性表达的过程。这一定义如此抽象，就是为了尽可能进行广泛的解释。“线性表达”可能是对一个句子，一个计算机程序，一组编织图案，一连串的地质地层，一首乐曲，礼仪仪式中的一组动作，总之线性序列就是前面的元素会以某种方式限制[1]下一个元素的表达。有一些语法已经被我们所掌握，而有些语法目前还在研究之中尚未完全确定，而还有一些语法目前仅仅只是有了一个大体的形状而内部完全不了解。

每种语法，一般都有无限的线性表达（“句子”）来构造表达。也就是说一组有限的语法结构，可以用来构造无限的句子。这是语法范式的主要力量，也是语法重要性的主要源泉：它简要的总结了某一个类的无数个对象的结构。

有几个理由可以展示这个被称为解析的构造过程。其中一个原因是，现有的结构能帮助我们更进一步的展示这个对象。比如当我们知道了一句话的某一个部分是这句话所描述的主体的时候，那么我们就更容易理解这一句话或者将这句话翻译出来了。一旦一篇文档的结构被弄明白之后，这篇文档就能很容易的被理解了。

第二个原因是，在某种意义上语法代表了我们对观察到的句子的理解：关于蜜蜂行为的解释，我们给出的解释越好则我们越能理解它们在做什么。

第三个原因是，部分丢失的信息，可以通过解析器，尤其是错误修复解析器（error-repairing parsers）来完善。如果能给出合理的语法，那么就能设计出一个错误修复解析器，来修复古阿卡德人的泥简（clay tablets）。

给定一堆句子，从中找出产生它们的语法结构，这种反向问题被称为文法推断。这个问题比解析技术，人们对其的了解要更少，不过仍然在进行当中。这个问题不在本书范围之内。国际文法推断座谈会（International Colloquiums on Grammatical Inference）的纪要由Springer出版，名为Lecture Notes in Artificial Intelligence。

> 1. 如果不存在元素之间的限制，序列仍然是有语法的，只不过语法也许十分繁杂并且难以理解。↩

1.1 解析技术

解析技术不再是一种神秘的艺术，自从上世纪70年代，Aho, Ullman, Knuth以及其他科学家为解析技术奠定的坚实的理论基础之后。解析技术也不是一门数学学科；一个解析器的内部工作过程可以是可视的、易理解的以及可修改的，用以适应应用程序，不再仅仅是简单的字符串剪切粘贴。

一位数学家的世界观和一位计算机科学家的世界观差距是相当巨大的。在数学家眼中所有的结构都是静态的：它们一直是静态的并且将永远都是静态的；只不过我们还没有发现全部的结构而已。而计算机科学家则倾向（并且很着迷）不断的创造、组合、分离和破坏结构：时间检验眞理。在数学家手里，Peano公理创建整数时没有提及到时间，但是如果一个计算机科学家使用这个公理来实现整数加法，那么他将发现这在计算机中是一个非常缓慢的执行过程，所以计算机科学家就会去寻找更加有效率的方法。在这方面，计算机科学家和物理学家和化学家拥有更多共同点；和他们一样，计算机科学家必须有数学的几个分支的坚实理论基础，但同时计算机科学家也很希望（而且经常迫使）数学家将一些理论的主导权交给他们，这点也和物理学家和化学家一样。没有严谨的数学理论的支持，所有的科学都将不复存在，但就像一栋大楼一样，并非其中的居民都需要了解这栋大楼的框架和梁柱。为各领域的专家们分化一些特殊的知识点的细节，可以减少复杂的脑力劳动，这是计算机科学家们正在做的事情。

本书是为这样的人准备的：需要做解析工作的人：解析器作者、语言学家、数据库接口作者、想要测试他们各自研究对象的语法的地理学家或音乐家、等等。我们需要读者有良好的可视化能力，一些编程经验以及跟进不太繁琐的例子的兴趣和耐心；了解一只袋鼠最好的方式莫过于亲眼去看它是如何跳跃的了。我们对流行的解析技术张开怀抱，同时我们也不会排斥一些奇怪的技术而只关注它们的理论知识：这些技术往往能为读者打开一扇新的大门。

1.2 使用方法

这本书的读者至少可以分为三个层次。感兴趣的非计算机科学家可以把这本书当作“语法和解析的故事书”；他们可以跳过解释算法的细节：只了解首次提到某个算法时的概要介绍。计算机科学家在研究各种算法时将会发现很多技术上的细节。我们为专家们提供了超过1700个项目的系统参考文献。本书的付印版本只包含了书中有引用的文献的参考目录；完整的参考目录收录在本书的网站上。付印版的全部参考文献和网站上将近三分之二的文献，都有注解；这些注解或者说摘要，与文中引用的内容无关，而是文献的简要说明，为了帮助读者判断这个文献是否值得一读。

本书给出了一些无法运行的算法，除了17.3中上下文无关的解析器。让程序员能实现并准确运行的解析算法公式，需要相当大的支持机制，而这不在本书研究的范围之内并且我们也没有足够的知识来为读者进行深入讲解。本书也给出了在大多数编译器构造书籍中多次介绍的算法。而那些不太为人所知的算法，则在原书中有详细介绍，第18章中列举了这些书籍。

1.3 内容概要

由于解析最关键的就是句子和语法，而语法本身又非常的复杂，所以第2章我们将对语法进行详细的讲解。第3章探讨了解析背后的原理，并给出了解析方法的分类。总之，解析技术可以分为自顶向下（top-down）或自底向上(bottom-up)两种，或者是定向(directional)和非定向(non-directional)两种；定向法又可以细分为确定性（deterministic）和非确定性（non-deterministic）的。这就决定了和紧接着后面几章的内容的主体。

第4章我们讲解非定向法，包括Unger和CYK。第5章介绍有限状态自动机（finite-state automata），为后面需要的章节做一个过渡。第6到10章介绍定向法，如下。第6章涵盖了非确定性的自顶向下解析器（向下递归，Definite Clause Grammars），第7章涵盖了非确定性的自底向上解析器（Earley）。第8章和第9章介绍确定性方法（第8章介绍自顶向下法：各种形式的LL。第9章介绍自底向上法：LR）。第10章涵盖非规范（non-canonical）的解析器，以一种不太规范的自顶向下或自底向上的方法来确定解析树的节点的解析器（例如left-corner）。第11章则介绍了类似上一章中的算法的非确定性版本（比如GLR解析器）。

接下来的四章内容，不太符合上述的框架。第12章介绍了最新的用于解析某一语言中完整句子的子字符串技术，包括确定性和非确定性的。第13章介绍了一种正在发展中的技术，这种技术将解析视为贯穿有限状态机的上下文无关语法。第14章介绍了几个并行解析算法，而第15章则解释了几种关于非Chomsky文法系统的建议以及他们的解析器。而这些本身就完成了解析方法。

第16章介绍了一些错误处理方法，第17章介绍了在写作和使用中比较实用的解析器。

1.4 参考文献

第18章是参考目录部分，在付印版和更加庞大的电子版中都是这样。它是本书主体部分的必要补充，也是容易获取的一个方式。参考文献被命名划分为几个小节，每一节都和解析有一定的关系，而不是通常的按照作者名字依次排列；付印版有25个小节而电子版有30个小节。每个小节内，文献的排列按照出版时间先后的顺序。书尾用文献的作者索引代替了通常的字母索引。文中方括号内的数字都指向一本参考文献。例如Earley写的关于Earley解析器的书在文中表示为[14]，那么在578页（原著的页数）被标记为14的那本书就是你要找的了。

2

2.1 语言，一个无限的集合

在计算机设备中，常规来说，“语法”是用来“描述”一门“语言”的。然而单从字面上来看的话，这样理解却会带来一些误解。计算机高手和初学者在看待这个问题上有一些细微的差别，具体在三个方面体现。为了建立和划定我们探讨的界限和范围，我们应该从刚才说到的三条差异来介绍，就从第三条开始吧！

2.1.1 语言

对大多数人类而言，语言，首要的作用是用来进行交流和思想沟通的。当人们在激烈的辩论中，这个作用尤为明显，不过当然了这是一种无意识行为。人们沟通交流，是指传递信息，更准确的说是通过空气振动传递声音信息或者是在纸面上书写文字信息。我们仔细看一下语言文字（话语），就是由一堆单词组成的句子，或者说，就是由一堆写在纸上的符号组成的信息。语言的组成上，总体来说有三个层面的区别。各种语言之间，有的存在微小的差异，就好比英语和爱尔兰语；但有的也存在巨大的差别，就好比英语和汉语。但是对单词的使用上却往往有这巨大的不同，甚至对使用同一种的人来说都是如此，比如在德语中“un cheval”和“ein Pferd”都表示“一匹马”。而句子结构的差异很容易被忽视，例如荷兰人中，有的人会有类似莎士比亚的语言习惯像“Ik geloof je niet”，如此来表达“我不相信你（I believe you not）”，而另一个地区的荷兰人则会有类似匈牙利的“Pénzem van”的语言习惯，像“Money-my is”表达的是英语中的“我有钱（I have money）”

但是，计算机研究学者却有一个非常抽象的角度来看待这一切。确实一种语言总是拥有着大量的句子，并且这些句子都有着一种特殊结构；我们不必关心这些句子是不是能传达某些信息，但句子们所具有的结构实实在在的含有了某种意义。而句子中包含的单词，这些在计算机中被称为“令牌（token）”的东西，每一个都包含了一些特殊的信息，而这些信息汇总起来，就是这个句子所被希望表达的意义。但是在自然语言中，词语是具有意义的不能再分的最小单位了，即使一个单词所包含的意义可能很多。不过，单词不能被更小的单位来表达对计算机研究学者来说并不是问题。通过可伸缩解决方案（telescoping solutions）以及多层技术（multi-level techniques），他们很容易的就能证明，如果一个词语确实含有某种结构，那么这个词就属于另外一种语言，一种把字母当作基础令牌的语言。

正式语言学的专家，通常被成为正式的语言学家（formal-linguist）（用以和“正式语言学家”做区分，这两者间的不同就留着读者见仁见智了）再次以一种不同的抽象角度来看待语言。语言是句子的“集合”，句子是“符号”的有

序集合；这就是说：如果一句话没有意义那是因为没有结构，一个句子是否属于某种语言，就看他的结构是否能在这种语言下产生意义。符号的唯一属性是它具有的标识;在任何语言中一定有一组确定数量的符号集，并且这一数字必须有限，比如字母表。比如，我们可以将这些符号写作a,b,c...，或者✆,✈,❒...也可以，只有数量是足够的就行。词语的顺序表明了，在每一个句子中词语都有他们固定的位置，我们不应该随意更改这些位置组合。而词语集则是把所有不同的词语都放在一起的一个无序的集合。这个无序集合可以写下来，通过把所包含的所有对象写在一对大括号中里面（就是这个{}）。对正式的语言学家（formal-linguist）来说，以下是一种语言：a,b,ab,ba；还有{a,aa,aaa,aaaa,...}也是一种语言，后者所涉及到的符号问题（逗号）我们稍后再说。根据计算机科学家眼中的“句子/单词，单词/字母”之间的对应关系，正式的语言学家（formal-linguist）也把句子称作一个“词语”，所以就会出现“ab这个句子，属于{a,b,ab,ba}这个语言”。

现在我们来思考一下这种巧妙而伟大的思想的含义。

对计算机研究学者来说，语言就是一个无限大的句子的集合，这些句子由令牌按照某种结构组成；令牌和特定结构通力合作最终表达了句子的语义，也就是句子的“含义”。变成语言的句子结构和语义都是全新的，也就是在现有的结构模式中所没有的，而研究学者们的任务就是提供和完善他们。对计算机研究学者来说，3+4×5 是“个位数运算”的一个句子（“个位数”则是为了保证数字是一个有限的符号集）；这个算式的结构可以这样展示：(3+(4×5))；而这个算式的含义就是 23.

即便是语言学家（linguist）也不得不承认，语言是所有可能有关联的句子的无限集合，这种说法要比上面的两种说法都要可接受的多。每一个句子由在现实生活中有明确意义的单词，按照结构化的方式组成。结构和词语一起传达出了一句话的意思。现在再次说到单词是由字母组成，这些字母和结构一起传达出了词语的意思。语义的重点、和现实世界的关系以及“句子/单词”“单词/字母”这两层的融合，是语言学家（linguist）的领域。“圆在疯狂的旋转”是一个句子，而“圆睡的发红”就是语无伦次了。

形式语言学家（formal-linguist）坚持他们对语言的看法，因为他们希望在研究语言的基本性质同时领略语言的原始美。计算机科学家坚持着自己的观点，因为他们想要一种清晰、 易于理解和明确的手段，描述计算机中的对象以及语言和计算机之间的通信，他们严格的不像人类。而语言学家（linguist）也坚持自己的看法，因为这让他找到了和一种看似杂乱无章并且似乎无限复杂的语言：自然语言，之间的紧密关联。

2.1.2 语法

每一个学习过外语的人都知道，语法就是一本充满规则和教会人们这门语言的示例的书。好的语法总是会仔细区分“句子/单词”和“单词/字母”的层级，前者通常被称为句法或句法集，后者通常被称为此法。句法包括一些规则，就像“pour que是虚拟语气，而parce que就不是”。词法也包括了一些规则比如“英文名词的复数形式一般词尾追加-s，除了单词本来就以-s.-sh,-o,-ch或-x结尾，这些单词为不规则复数形式，结尾追加-es”。

我们暂时先跳过计算机科学家看待语法的视线，先看看正式语言学家（formal-linguist）眼中的语法。正式语言学家的观点是抽象的同时又和外行的看法很相似：语法是任何语言的确定的、有限大小的、完整的描述，即句子的集合。这实际上是学院派的语法，只是去掉了其中的模糊性。虽然很明显这一定义具有充分的通用性，但却又太过笼统以至于显得很无力。它包括一些描述像“可能是乔叟写的一组句子”；柏拉图谈到这定义了一个集合，但是我们却没有办法创建这个集合或度量一个句子是否属于这种语言。This particular example, with its “could have been” does not worry the formal-linguist, but there are examples closer to his home that do.“π的超长的小数部分是一个最长的块”描述了一种语言，这种语言最多只有一个单词（这个单词就是乱七八糟的数字），而且很符合定义说的精确、有限大小并且完备。然而又有一个问题，那就是没人知道这个词完整的样子：假设即便有一个人找到了一堆长度为数十亿的数字，在这后面却任然还有着数不尽的数字。另一个问题就是，我们还根本无法知道这么长的数字是不是真的存在。很可能就是一个人在不停的发现π的小数，另一个人却发现了越来越长的小数仍然没有被找到。关于π的小数的全面展开理论也许能解释这些问题，不过目前这个理论还不存在。

因为一些这样或那样的原因，正式语言学家们放弃了他们静止的柏拉图式的观点，转而接受了另一个更有建设意义的，衍生语法：衍生语法是语言中构建句子的准确且固定大小的诀窍。这意味着，使用这个语法就一定可以构建语言（动作数量有限）中的句子，而不会有其他的可能。这并不是

说，给一个句子，语法就能让我们知道这个句子是如何构造的，只是说我们可以通过语法来知道。这些语法可能有几种形式，其中有一些会比其他更具便捷性。

计算机科学家们通常会赞同这个的观点，而且还有一个附加的要求就是衍生语法应该隐含一句话是如何构造的。

2.1.3 无限集带来的问题

上述将语言定义为序号序列的无限集，以及将语法定义为构造句子的配方的定义，引出了两个令人尴尬的问题：

1.一个有限的配方如何能产生无限的句子集呢？

2.如果一个句子只是一个序列而没有结构，或者如果一个句子，可以通过其结构推导出其他的意义，那我们该如何理解这个句子呢？

这两个问题有一个漫长而复杂的答案，不过确实是有答案的。我们先解决第一个问题，然后在带着第二个问题去阅读本书的主体部分。

2.1.3.1 有限描述的无限集

其实从一个有限的描述中得到一个无限集，并没有什么问题："所有正整数集合"是一个非常有限的描述，但描述的却是一个无限集合。不过，还是有一些令人不安的想法，所以我们把问题换一个说法："所有的语言都能用有限的描述来说明吗？" 正如上文暗示的，答案是"不行"，不过证据却不是无关紧要的。实际上是非常有趣并且有名气的，如果不展示一下或者至少大致的介绍一下，将会是一个遗憾。

2.1.3.2 枚举描述

证明是基于两个意见和一个技巧之上的。第一个意见是，描述是可以列举的并且有一个数据。如下所示。首先，找出全部大小为1的，也就是那些上都只有一个字母的，然后将他们按字母顺序排序。这是我们列表的开头。基于这个，实际上我们在接受一种描述，长度为1的描述可能是0，或27（所有字母+空格），或者95（所有可打印的ASCII字符）或者其他类似的；具体是什么对下面的讲解都不重要。

第二步，我们找出长度为2的并将他们按照字母顺序排序，然后将他们放在列表中的第二块；然后是长度为3的，长度为4的等等等等，都这样做。每一个描述都这样在列表中获得一个位置。例如“所有正整数的集合（the set of all positive integers）”这个描述，集合大小为32个字符（英文），不算引号。若要查找其在列表中的位置，我们要先计算有多少少于32字符的描述，称为L。然后我们必须生成所有长度为32的描述，对他们进行排序，然后确定我们的描述在其中的位置，称为P，然后把L和P加起来。这肯定会是一个巨大的数字[1]，不过这就能确保我们的描述处于这个完整定义的列表中了；见图Fig.2.1。

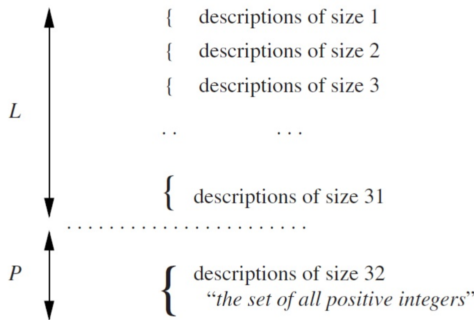


Fig. 2.1. List of all descriptions of length 32 or less

有两件事情应该指出。第一，只是根据字母列出全部描述，而不指出其长度将不会起作用：已经有很多以“a”开头的描述，以及没有以其他字母开头的描述，将会在列表中获取一个位置。第二，其实没有必要列出全部的描述。这只是一个思想实验，使我们在无法亲自检测结构的情况下，能有一个方式来检测一个行为体系并得出结论。

此外，列表中会有一些荒谬的描述；不过这对我们的论点来说是无关紧要的。重点是有用的描述都在列表中，以上描述能确保这一点。

> 1. 一些计算表明，在ASCII-128假设下，这个数字是248 17168 89636 37891 49073 14874 06454 89259 38844 52556 26245 57755 89193 30291，或者大致是2.5×10^{67}。↩

2.1.3.3 语言，无限的比特串

我们知道，在语言中单词（句子）被视为一组有限的符号集；这个集合被称为“字母表”。我们假设字母表中的字母是有序的。那么语言中的单词也是有序的。我们用字母Σ来表示字母表。

现在最简单的语言就是，使用字母表Σ中的字母组成了所有单词的语言。通过字母表Σ = {a,b}，我们获得了一门语言{ , a, b, aa, ab, ba, bb, aaa, . . . }。我们应该把这个语言称为Σ^*，稍后再说为什么这么称呼；暂时这只是一个名字。

上面用Σ^*标记的集合，以“ { , a,”为开头，很明显的一个结构；这个语言的第一个单词是一个空的单词，这个单词由A集的大小为0的和B集的大小为0的单词构成。没有理由拒绝这个空单词的到来，但如果写下来又很容易被忽视掉，所以我们把这个空单词写成ε(小写的Σ)，不理睬字母表。所以，Σ^* = { ε, a, b, aa, ab, ba, bb, aaa, . . . }。在一些自然语言中，动词“to be”的现在时态就是一个空单词，赋予“I student”这个句子以“I am a student”的意义。俄语和希伯来语就是很好的例子。

因为字母表Σ中的符号是有顺序的，我们可以列出语言Σ^*中的单词，使用上一节中说到的方法：第一，将所有大小为0的单词排序；然后大小为1的单词排序；然后后面依次。实际上我们已经在Σ^*中这样做了。

语言Σ^*有一个有趣的地方，就是所有使用字母表Σ的语言，都是它的子集。这意味着，在Σ的基础上创建一门不那么复杂的语言，称做L，那我们就能遍历Σ^*列表中的所有单词，然后在属于L的单词上做一个标记。这样就能把L中的全部单词都包括了，因为Σ^*本来就包括了所有可能属于L的单词。

假设我们的L语言是“一组相比于B集，包含更多A集的单词的集合”。L就是这样的{a, aa, aab, aba, baa, . . . }。那Σ^*列表的开头部分就是下面这样的：

	ε
✔	*a*
	b
✔	*aa*
	ab
	ba
	bb
✔	*aaa*
✔	*aab*
✔	*aba*
	abb
✔	*baa*
	bab
	bba
	bbb
✔	*aaaa*
···	···

给定一个排序过的字母表，空格和有标记的部分就完全足够识别和描述一个语言了。为了方便，我们将空格写作0，将有标记的写作1，就像计算机中的位（bits）一样，那我们现在就可以把L写成L = 0101000111010001· · ·（而Σ^* = 1111111111111111· · ·）。需要指出的是这适用与任何语言，比如一门正式语言L，一门编程语言Java，一门自然语言英语。在英语中，像标记为“1”这样的是非常稀少的，因为几乎没有任何一个任意序列的单词组合会是一个有意义的句子（同理，没有任何一个任意序列的字母组合是一个有意义的单词，取决于我们怎么处理“句子/单词”和“单词/字母”这两个层面）。

2.1.3.4 对角化

上一节将无限位字符串0101000111010001· · ·和“一组相比于B集，包含更多A集的单词的集合”这个描述联系在一起。同样，我们可以将这种位字符串附加到所有的描述上。有些描述可能不足以产生一门语言，这种情况下我们可以将任意无限位字符串附加给它。既然所有的描述都可以有一个单一的被编号的列表，那么我们就有了下面的图：

Description	Language
Description #1	000000100···
Description #2	110010001···
Description #3	011011010···
Description #4	110011010···
Description #5	100000011···
Description #6	111011011···
···	···

左侧是描述，右侧是描述对应的语言。但是现在我们说很多已经存在的语言却不存在于上述列表中：上述列表离完整还很远，虽然列表的描述是很完善了。 为了证明这点，我们将使用Cantor的对角化过程（“Diagonalverfahren”）。

想象一下，语言C = 100110···，它的第n阶位并不等于描述#n中描述的第n阶位。语言C的第一个比特位为1，因为描述#1的第一个比特位是0；第二位是0，因为描述#2的第二位是1，等等。C是由语言字段的对角线的左上方到对角线的右下方，然后复制我们遇到的每一个位的反向位组成。这就是图Fig 2.2（a）中对角线。那语言C就不能存在于列表中！它不能在行1中，因为它的第一位不同于（应该说，被弄得不同）行1的第一位，而且通常来说，它也不能在行n中，因为第n位不同于行n的第n位，通过定义可以得出。

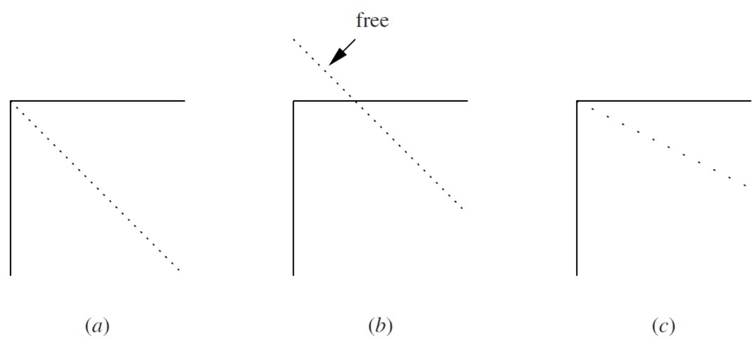


Fig. 2.2. “Diagonal” languages along n (a), $n+10$ (b), and $2n$ (c)

所以，尽管我们已经详尽无遗的列举了所有可能的有限描述，那也至少会有一种语言的描述不在列表中。但实际上，有更多的语言不在列表中。例如，构造一个语言，它的第n+10位于描述#n的第n+10位不同。这个语言再

次不存在于列表中因为每一个n>0的位都不同于行n中第n+10位。但是这意味着1到9位没起作用，可以任意替代，如图Fig 2.2（b）所示；这将生成另外2^9 = 512种语言，且都不在列表种。而且我们能做的更好！假设我们构建了一门语言，它的第2n位与描述#n（c）的第2 n位不同。明显它也不在原列表中，但现在每一个基数位都是左侧未指定的部分，而且可以随意选择！这使得我们可以自由的创建无限数量的语言，而它们都没有有限的描述；见图Fig 2.2中的斜对角线。简而言之，对于每一种可以描述的语言来说，还有更多的不可描述的语言。

关于对角化技术，在理论计算机科学中的很多书中有更正式的讲解；例如Rayward-Smith [393, pp. 5-6], 或者 Sudkamp [397, Section 1.4]。

2.1.3.5 讨论

上面的论证展示给我们几件事情。第一，这表明了把语言当作一个正式对象的力量。虽然上述大纲明显还需要相当大的扩增和证据，来证明其资质（为了这个，却仍需加以澄清）为什么上述定义语言C的解释，本身不在描述列表中；见问题2.1，这让我们能深入了解其属性，或者评估其属性。

第二，这表明我们只能描述所有可能语言的很微小的一部分子集（甚至都不是一小部分）：语言的数量是无限的，且远远超出我们能弄清的范围。

第三，我们已经证明，虽然有无穷多的描述和无限多的语言，然而描述和语言的范围却并不相等，后者要远大于前者。这些无穷被Cantor称为$\aleph_0$和$\aleph_1$，而上述只是他对$\aleph_0 < \aleph_1$进行证明的一个特殊例子。

2.1.4 通过有限集描述一门语言

一个好的生成一组对象的方法，是先创建一个小对象，然后制定根据现有对象添加新对象的规则来生成一组完整的对象。“有两个偶数，这两个偶数相加得到的数也是偶数”这样就很有效的生成了所有的偶数的集合。 形式主义者会添加一个说明“并且没有别的数是偶数”，不过我们都理解这个。

假设我们想要生成所有人名的枚举的集合，就像“Tom，Dick和Harry”这种，集合里面除了最后两个之外，其他都用逗号分隔开来。我们不接受“Tom，Dick，Harry”或者“Tom和Dick和Harry”这种类型的，但是我们接受重复的，比如“Grubb, Grubb和Burrowes”[1]是可以的。虽然在英语中这不是完整句子，可我们依旧要把他们称作“句子”，因为它们是我们的微型语言——人名枚举集——中的句子。这个简单的结构是：

1. Tom是一个人名，Dick是一个人名，Harry也是一个人名；
2. 一个人名就是一个句子；
3. 一个句子后面跟着一个逗号，然后后面跟着一个是句子的人名；
4. 在枚举结束之前，如果句子的结尾是“，人名”，那用“和人名”来替换。

虽然这样说对于读者的理解很管作用，但是却有几个错误的地方。条件3尤其会带来麻烦。例如，句子并不是以“，人名”结束，而是以“，Dick”或类似的名字结束，“人名”只是一个符号用以代替真是的名字；这样的符号不能在真是的句子中出现而且在结尾处的必须用条件0中给出的名字替代。同样，条款中的“句子”是一个符号，指代真实的句子。所以这里涉及到两种符号：出现在完备句子中的真实符号，比如“Tom”，“Dick”，逗号和“和”；还有中间体符号，像“句子”和“人名”这些不能在完备句子中出现的词。第一种对应前文中提到的单词或令牌；它们的专业术语是终结符（或简称终结）。这种中间体符号被称为非终结符，一个很平庸的术语。为了区分它们，我们将终结符用小写字母表示，非终结符用大写字母表示。 非终结符被称为（语法的）变量或句法的类，在语言学语境上。

为了强调按照条件生成的字符，我们应该将“X就是Y”替换成“Y应该用X替换”：如果“tom”是人名（Name）的一个实例，那么不管我们在哪儿看到人名（Name）我们就可以将之缩小范围为“tom”。这给了我们：

1. 人名（Name）可能被“tom”取代

   人名（Name）可能被“dick”取代

   人名（Name）可能被“harry”取代

2. 句子（Sentence）可能被人名（Name）取代
3. 句子（Sentence）可能被句子（Sentence），人名（Name）取代
4. 句子（Sentence）结尾处的“, 人名（Name）”必须被“和人名（Name）”取代，在人名（Name）被任何替代物替代之前
5. 一个句子只有当它不在包含非终结符时才能结束
6. 我们从句子（Sentence）开始替换过程

条件0到3说明替代物，但是4和5不一样。条件4不是特定于此语法的。它基本都是有效的，而且是我们这个游戏（解析）的规则之一。条件5告诉我们从哪里开始生成。它的名称自然而然的被称为起始符，而且它是每个语法都要求有的。

条件3还是看起来不让人放心；其他大多数规则都写着“可能被取代”，而这条写着“必须被取代”，并且它是指“一个句子的结尾”。其他规则通过替换来生效，但是问题仍在，我们如何用替换来检测一个句子（Sentence）的结尾。这个问题可以通过在结尾处添加一个终止符来解决。如果我们对一个非终结符做了终止标记，这个非终结符不能在除了从“，人名（Name）”替换为“和人名（Name）”处之外使用，那我们就自动执行了这个限制——除非替换检测已经发生否则没有句子将会结束。为了简便起见我们写作——>而不是“可能被替代”；由于终结符和非终结符现在是专业术语，所以我们把他们写成打字机的字样。——>之前的部分称为左手边，之后的部分称为右手边。这就成了图Fig 2.3中的样子。

```
0.          Name  →  tom
            Name  →  dick
            Name  →  harry
1.      Sentence  →  Name
        Sentence  →  List End
2.          List  →  Name
            List  →  List , Name
3.    , Name End  →  and Name
4.  the start symbol is Sentence
```

Fig. 2.3. A finite recipe for generating strings in the t, d & h language

这是这个配方的简单和比较精确的形式，规则也很简洁明了：由起始符开始，持续替换过程直到没有非终结符剩下。

> 1. 引用至The Hobbit, by J.R.R. Tolkien, Allen and Unwin, 1961, p. 311. ↩

2.2 正式语法

上述的配方形式，基于根据规则的替换，已经强大到足以支撑正式语法的基础了。相似的形式，通常被称为“重写系统”，在数学家中已经有了悠久的历史，而且在公元前几个世纪已经在印度被使用了（例如，见Bhate and Kak [411]）。图Fig 2.3第一次被广泛研究是由Chomsky[385]做的。他的研究成果成为了正式语言、解析器和相当大部分的编译器构造以及语言学的几乎所有研究和进展的基础。

2.2.1 正式语法的形式主义

由于正式语言是数学的一个分支，所以这一领域的工作是在一个特殊符号下完成的。若要展示一些他的韵味，我们就应该正式定义一下语法，然后解释一下为什么这样就描述了一种语法，就像图 Fig 2.3中的那个一样。形式主义的使用是必不可少的，为了证明的准确性等，但是不是为了理解其基本原理；在这展示只是为了给读者一个印象，或者说跨过沟壑的一个桥梁。

定义**2.1**: 一个生成语法是一个4元组（V_N,V_T,R,S），像这样：

(1) V_N和V_T是符号的有限集，

(2) $V_N \square V_T = \square$,

(3) R是一个有序数对（P,Q），像这样

(3a) $P \square (V_N \square V_T)^+$ 和

(3b) $Q \square (V_N \square V_T)^*$,

(4) $S \square V_N$

一个*4*元组只是一个包含了4个可辨识部分的对象；这4个部分按照顺序是非终结符、终结符、规则和起始符。上述定义没有说明这点所以这需要老师来解释。非终结符集用V_N来表示，终结符集用V_T来表示。对我们的语法来说，我们就有

V_N = {人名，句子，列表，结尾}（{Name, Sentence, List, End}）

V_T = {tom, dick, harry, ,, and}

（注意，终止符的那个集合）

V_N和V_T（2）的交集必须是空集，用空集合符号 □ 表示。所以非终结符集和终结符集不可能有公共元素，这是可以理解的。

*R*是所有规则（3）的集合，*P*和*Q*分别是放左侧和放右侧的内容。每一个*P*必须包含一个或多个非终结符和终结符的序列，每一个*Q*必须包含零个或多个非终结符和终结符的序列。对我们的语法来说，我们就有：

R = {**(Name, tom), (Name, dick), (Name, harry),(Sentence, Name), (Sentence, List End), (List, Name),(List, List , Name), (, Name End, and Name)**}

再次注意，两个不同的逗号。

起始符*S*必须是V_N里面的元素，也就是说必须是非终结符：

S = **Sentence**

我们的领域在这里就结束了，后面是语言学的领域。简言之，数学上的正是语言就是一门语言，一门必学的语言；它让表达“是什么”和“怎么做”变得非常简单，但是关于“为什么”却给了很少的说明。把这本书当作一个翻译和一个注释。

2.2.2 通过正式语法生成句子

图Fig 2.3中的语法，就是所谓的我们的**t,d&h** 短语结构语法（通常缩写为PS语法）。还有一个更紧凑的形式，左侧相同的，对应的右侧写在一起用竖线“|”隔开。这条竖线只是一种形式，就像箭头--->一样，可以读作“或其他”。右侧被竖线分隔开的，也被叫做备选项。这种更简洁的形式下，我们的语法变成了：

0.	Name	→	tom | dick | harry
1.	Sentence_S	→	Name | List End
2.	List	→	Name | Name , List
3.	, Name End	→	and Name

带有下角标$_S$的非终结符成为了起始符。（下角标决定了起始符，而不是规则）

现在让我们用这个语法来生成我们的初始例子，只使用根据上述规则的替换方式。我们得到了以下连续形式的句子：

Intermediate form	Rule used	Explanation
Sentence		the start symbol
List End	Sentence → List End	rule 1
Name , List End	List → Name , List	rule 2
Name , Name , List End	List → Name , List	rule 2
Name , Name , Name End	List → Name	rule 2
Name , Name and Name	, Name End → and Name	rule 3
tom , dick and harry		rule 0, three times

中间形式（intermediate forms ）被称为句子形式。如果一个句子形式不包含非终结符，那它则被称为一个句子并且属于生成的语言。从一行到下一行的过渡被称为生成步骤，而其规则被称为生成规则，原因很明显。

生成过程可以更加直观，通过使用“图标”在对应符号之间画一条连接线。一个图就是一个节点集合，和一组边连接起来的。一个节点可以认为是纸上的一个点，一条边是一条线，每条线连着两个点；一个点可能是多条线的结束点。图中的节点通常都是被“标记”的，也就是说它们都有名字，这样就

可以很方便的在纸上画出写着它们名字的小圈来代替这些点。 如果这些边是有箭头的，那这个图就是有方向的；如果这些边只是线条，那这个图就是无方向的。几乎所有在解析技术中使用的图都是有方向的。

图标对应上述生成过程见图Fig 2.4。这种图被称为生成树或者句法树，描述了最终句子的句法结构（在给定的语法中）。我们看到的生成图通常是向下的，但偶尔我们也会看到一些星型结构，一般是由重写了一组符号导致。

在图中的一个循环就是一个路劲，从节点N顺着箭头一直在回到节点N。而生成树中是不允许有环的；如下所示。要得到一个环，我们就需要在生成树中有一个非终结符节点N，且N已经有了一个直接或间接指向N的子节点。但由于生成过程总会产生节点的副本，所以他布恩那个生成已经存在的节点。因此一个生成树是“非循环”的；有向无环树被称为*dags*。

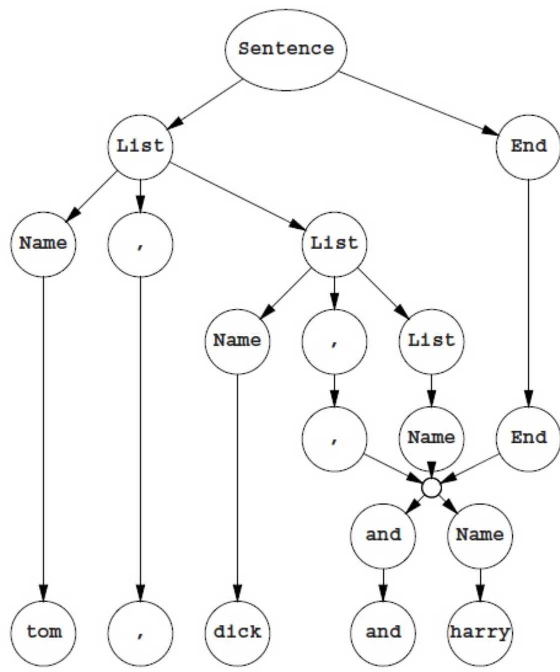


Fig. 2.4. Production graph for a sentence

明显不可能让语法生成tom，dick，harry，任何试图生成多个人名的企图都会带来结束，而摆脱它（我们也必须摆脱，因为这是一个非终结符）的唯一办法就是把第3条规则吸收进来，而这会产生一个“和”。神奇的是，在

一个只使用"可能被替换"的系统中我们成功的实施了"必须替换"这一概念；仔细看一下，我们就会发现，我们将"必须替换"分成了"可能替换"和"必须不能是一个非终结符"。

除了我们的标准例子，语法当然还会生成一些其他的句子；例如：

harry and tom harry tom, tom, tom and tom

以及其他更多的。一个决绝和鲁莽的尝试，生成没有"和"的不正确的形式将会让我们得到这样的句子形式，像

tom, dick, harry End

这样的，不是句子的句子形式，而其没有生成规则适用于它们。这种形式被称为死胡同。就像生成规则中的右箭头表明的一样，在相反的方向上规则可能就不适用了。

2.2.3 正式语法的表达力量

正式语法的主要属性就是它有生成规则，可用于重写部分句子形式（=正在构建的句子），和产生所有句子形式的起始符。在生成规则中我们发现有终结符和非终结符；而在完整的句子中只有终结符。这大概就是：剩下的部分就看语法作家和句子创造者的创造力了。

这是一个令人印象深刻的简洁的框架，然后就马上有了一个问题：这就足够了吗？这很难说，但是如果这还不是最好的，那我们暂时也没有更好的表达形式了。听起来可能有点奇怪，所有其他人们已知的生成集合，都被证明相同或甚至不那么强大，比起短语结构语法。明显生成一个集合的方法，当然最简单的是写一个程序来生成，不过这也证明了所有可以用程序生成的集合都可以用短语结构语法来生成。甚至还有一些神秘的方法，但最终都被证明不够具有表现力。另一方面，也没有证据能表明还存在更强大的方法。但鉴于许多完全不同的方法最后都由于相同的阻碍而停止，所以几乎不可能[1]在找到一个更强大的方法了。见例子Révész [394, pp 100-102].

作为表现力的另外一个例子，我们应该谈谈Manhattan龟的运动的语法。Manhattan龟只能在平面上运动，且只能往东南西北方向运动，每次运动距离为一个格子。图Fig 2.5给出了所有可以回到原点的循环路劲。根据第2条规则，应该说明，许多作者需要在左侧至少有一个非终结符的符号。这条规则总是可以强制执行通过添加新的非终结符。

```
1.          Moves  →  north Move south | east Move west | ε
2.     north east  →  east north
      north south  →  south north
       north west  →  west north
       east north  →  north east
       east south  →  south east
        east west  →  west east
      south north  →  north south
       south east  →  east south
       south west  →  west south
       west north  →  north west
        west east  →  east west
       west south  →  south west
```

Fig. 2.5. Grammar for the movements of a Manhattan turtle

简单的东南西北循环路劲已经在图Fig 2.6中展示（使用首字母缩写）。注意规则（ε）中的空替代，这导致了上图中第三个M后的结束。

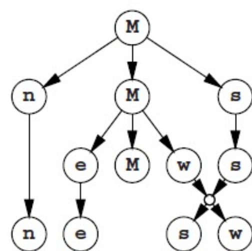


Fig. 2.6. How the grammar of Figure 2.5 produces a round trip

> 1. Paul Vitány指出如果科学家说神秘东西“几乎不可能” has pointed out that if scientists call something “highly unlikely” they are still generally not willing to bet a year’s salary on it, double or quit. ↩

2.3 语法和语言的Chomsky层次结构

图Fig 2.3和2.5中的语法都很容易理解，而事实上一些简单的短语结构语法可以生成复杂的集合。然而，对于任何一组给定的语法通常都很难说是简单的。（虽然语法有无限的集合，但是在这我们还是说“一个给定集合的语法”。通过一个集合的语法，我们来解释任何做了同样工作的语法，并且不是过于复杂的。）理论上说如果一个集合最终可以生成（例如，通过一个程序），那么它就可以通过一个短语结构语法来生成，不过理论上没有表明这样做很容易，或者说语法是容易理解的。在这一节中要说明Manhattan龟的路劲尽量写出其语法，这些路劲中龟总是不能从起始点向西爬行的。（问题2.3）。

不考虑短语结构语法带来的智力问题，他们也表现出了基本和实际的问题。我们可以看出对这些集合没有普适的解析算法，而所有已知的特殊算法，要么效率很低要么就是很复杂；见3.4.2节。

限制短语结构语法超出控制的想法，也就是尽可能的保持它们的生成力，导致了语法的Chomsky层次结构。这种层次结构分为了四种类型的语法，编号为0到3；引入第5种类型，也就是被称为类型4的类型，是有意义的。0型语法是（无限制的）短语结构语法，我们已经看到了它的例子。其他类型源于越来越被限制的语法规则中被允许的形式。每一种限制条件都带来深远的影响；最终语法逐渐变得更易于理解和掌握，但也逐渐不那么强大。幸运的是，这些不那么强大的语法依旧十分有用，甚至实际上比0型更有用。

我们现在先按顺序来看看前三个类型，然后接着在看看琐碎但有用的第四个类型。

生成0型语言的完全不同的方法，见示例Geffert[395]。

2.3.1 1型语法

0型语法的特征是，它包含一条规则是可能将任意数目（非零）的符号转化为任意数目（可能为零）的符号。例如：

, N E ---> and N

如上三个符号被两个符号取代。通过限制这种自由，我们引入1型语法。奇怪的是关于1型语法的，直觉上完全不同的两个定义，却很容易被证明是相同的。

2.3.1.1 1型语法的两个类型

一种语法是*1*型单调，如果它不包含规则，左侧的符号比右侧符号更多。例如，禁止这个规则 **, N E ---> and N**.

一种语法是*1*型上下文相关，如果所有的规则都是上下文相关。一个规则是上下文相关，如果左侧只有一个符号（非终结符）被其他符号替代，然后我们把右侧的符号都找到且没有损坏且按顺序放置。例如：

Name Comma Name End ---> Name and Name End

这就说明这个规则

Comma ---> and

可能适用，如果左侧文字是**Name**而右侧文字是**Name End**。上下文本身不受影响。替代者必须是至少一个符号长。这意味着上下文相关语法总是单调的；见2.5节。

这里是**t,d&h**例子的一个单调语法。在书写单调语法时，必须要小心绝不能多生成一个后面本来就会生成的符号。我们通过将结束标记纳入最右侧的名字来避免需要删除键鼠标记：

```
           Name  →  tom | dick | harry
      Sentence_s  →  Name | List
           List  →  EndName | Name , List
      , EndName  →  and Name
```

当**EndName**是一个单独的符号时。

这就是它的一个上下文相关语法。

```
           Name  →  tom | dick | harry
      Sentence_s  →  Name | List
           List  →  EndName
                    | Name Comma List
  Comma EndName  →  and EndName         context is ... EndName
    and EndName  →  and Name            context is and ...
          Comma  →  ,
```

请注意，我们需要一个额外的非终结符逗号来产生终结符，并在正确的上下文中。

单调语法和上下文相关语法时同样强大的：每一种可以由单调语法生成的语言，都可以由上下文相关语法生成，反之亦然。但它们都不如0型语法强大，也就是说，可以由0型语法可以生成任何一种1型语法都无法生成的语言。但奇怪的是没有这样的语言被人们所知道。虽然0型语法和1型语法的区别是基础层面的，而且不是Chomsky先生一时兴起的，它们之间差异的语法太复杂而不能写下来；而只能证明这差异的存在（例如Hopcroft and Ullman [391, pp. 183-184], 或 Révész [394, p. 98]）。

当然任何类型的1型语法都属于0型语法，因为1型语法是在0型语法的基础上，增加了一些限制条件推导而来。但如果将1型语法也称为0型语法就会引起混乱；就像把猫叫做哺乳动物一样：正确但不精确。一种语法是按照最小适用类型来命名的（也就是，最高类型数）。

我们可以看到语言**t,d&h**，先由0型语法生成，也可以由1型语法生成。我们应当也能知道同样还有2型和3型语法适用，但没有4型语法。所以我们说语言**t,d&h**是一种3型语言，经过限制层数最多（最简单和适合）的语法之后。关于此的一些推论：一个n型语言可以由n型语法或更强大的语法生

成，但不能由比较弱小的n+1语法生成；以及：如果一种语言是由n型语法生成，那并不意味着（更弱小的）n+1型语法无法生成它。**t,d&h**语言的0型语法是一个严格的例子，而只是为了演示目的。

2.3.1.2 构建一个1型语法

1型语法的标准例子就是包含相同数目的a,b,c的集合，按照以下顺序：

$$\underbrace{a\,a\,.\,.\,.\,.\,a}_{n\text{ of them}}\quad\underbrace{b\,b\,.\,.\,.\,.\,b}_{n\text{ of them}}\quad\underbrace{c\,c\,.\,.\,.\,.\,c}_{n\text{ of them}}$$

为了完整起见，以及展示1型语法如何被书写出来，如果够聪明的话，我们现在应该得出这个玩具语言的语法。从最简单的例子开始，我们有了以下规则：

0、**S ---> abc**

获得了S的一个实例，我们可能会想要在开头追加更多的a；如果我们想要记住已经有多少了，那同时我们就应该附加一点东西在结尾处，而且不能是一个b或者c。我们应该使用一个目前未知的符号Q。追加后的规则如下：

1、**S ---> aSQ**

如果我们用该规则，例如，使用三次，我们就得到下列句子形式：

aaabcQQ

现在，若要从这中间获得**aaabbbccc**，每一个Q必须等同于一个b和一个c，如预期，但我们不能这么写：

Q ---> bc

因为这会导致第一个c后面是b。因此，如果限定更换只允许在左边是b右边是c的情况下发生，那上述规则就是可行的。新插入的bc就不会造成影响，如下：

2、bQc ---> bbcc

不过，我们不能应用此规则，因为通常Q是在c的右边。可以通过允许Q向左跳过一个c来纠正：

3、cQ ---> Qc

现在，我们可以完成我们的推导：

```
aaabcQQ     (3 times rule 1)
aaabQcQ     (rule 3)
aaabbccQ    (rule 2)
aaabbcQc    (rule 3)
aaabbQcc    (rule 3)
aaabbbccc   (rule 2)
```

应该指出的是，上述推导只显示语法能生成正确的字符串，读者需要说服自己，语法不会生成别的东西或不正确的字符串（问题2.4）。

语法总结为了图Fig 2.7。因为$a^3b^3c^3$的推到图已经相当笨拙，

1.	S_s	→	abc	aSQ
2.	bQc	→	bbcc	
3.	cQ	→	Qc	

Fig. 2.7. Monotonic grammar for $a^n b^n c^n$

所以图Fig 2.8给出了$a^2b^2c^2$的推到图。

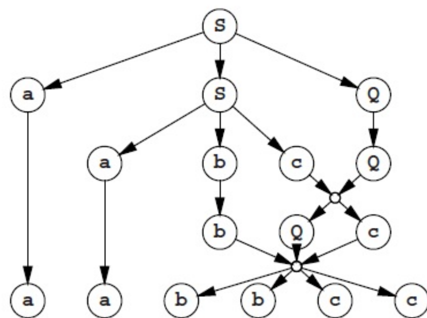


Fig. 2.8. Derivation of aabbcc

语法是单调的，因此属于1型。可以证明这个语言不符合2型；见2.7.1节。

虽然只有上下文相关的1型语法拥有被称为上下文相关语法（CS语法）的权利，但这个名字经常被使用在单调1型语法上。对于单调语法没有标准缩写，但MT可以用来代替。

2.3.2 2型语法

2型语法被称为上下文无关语法（CF语法），其和上下文相关语法之间的关系如它们的名字所示。上下文无关语法和上下文相关语法是相似的，只是左侧和右侧的文字必须是空缺的（空的）。因此语法可能只包含一个规则，在其左侧只有一个非终结符。语法示例：

```
0.          Name  →  tom | dick | harry
1.  Sentence_s    →  Name | List and Name
2.          List  →  Name , List | Name
```

2.3.2.1 独立生成

由于左侧总是只有一个符号，生成图中的每个节点具有这样的属性，无论其生成什么都独立于其相邻节点的生成：非终结符的生成周期独立于其上下文。我们在图2.4，2.6，2.8中见到的星型结构不可能发生在上下文无关语法中，因此就会有一个纯粹的树形结构，其被称为生成树。示例如图2.9。

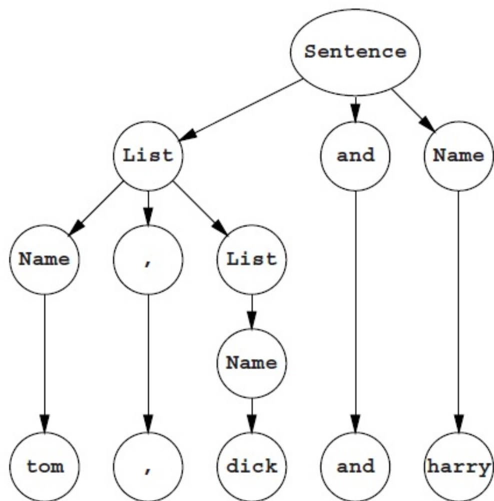


Fig. 2.9. Production tree for a context-free grammar

由于左侧只有一个符号，对于一个给定非终结符的所有右侧，可以放在同一个语法规则的集合中（在上述语法中我们已经做了），然后每个语法规则就像一个左侧的定义：

- 一个**Sentence**是后面接着**and**接着**Name**的**Name**或是**List**.
- 一个**List**是**Name**后面接着一个a，或接着**List**或是**Name**.

这显示了上下文无关语法构建字符串是经过了两个过程的：拼接（“...接着...”）和选择("要么...要么....")。除了这些过程在这里是识别机制，其链接了在右侧使用的非终结符的名字和定义规则（“...是一个...”）。

在这章节的开头，我们确定了一门语言作为字符串的一个集合，起始符号的终结符的生成物的集合。独立生成物的属性让我们可以把这个定义扩展到这个语法的任何非终结符上：每一个非终结符都生成一个集合、一门语言，独立于其他非终结符生成的。如果我们将A生成的字符串集合写作$L(A)$，而且A有两种可选的生成规则，$A \rightarrow \alpha|\beta$，那么$L(A) = L(\alpha)$ □ $L(\beta)$，□代表集合的并集运算符。这对应于上一段说到的选择。如果α由PqR三个字符组成，我们就有$L(\alpha) = L(P)$ □ $L(q)$ □ $L(R)$，□代表字符串的串联运算符（实际上上集合中的字符串）。这对应于上一段说到的拼接。而$L(a)$中的a是终结符，属于集合{a}。语言中包含ε的非终结符被称为空。也可以说“生成空”。

注意，我们不能定义图Fig 2.7中Q的语言$L(Q)$：Q本身不能生成任何有意义的东西。为非起始符号定义语言只能在2型语法或更低级的语法上可能实现，将非起始的非终结符延议为空也是一样的。

有关独立生成属性就是递归的概念。非终结符A是递归的，如果句子形式中的这个A可以生成再次包含A的东西。图Fig 2.9中的生成物开始于句子形式的Sentence，其使用了规则1.2来生成List和Name。下一步很可能是用Name,List来替换List，使用规则2.1。我们看到List生成了再次包含List的东西：

Sentence ---> List and Name ---> Name , List and Name

即List是递归的，尤其是，它是直接递归。非终结符A在**A--->Bc, B--->dA**中是间接递归的，不过这之间的差异并没有太多意义。

比之更重要的是List是右递归的：一个非终结符A是右递归的，如果它可以在右侧生成包含A的东西，则List为：

List ---> Name , List

同理，一个非终结符A是左递归的，如果它可以在左侧生成包含A的东西：我们就可以定义

List ---> List , Name

一个非终结符A是自嵌入的，有这样一个定义：如果A能在生成一个，左侧是α右侧是β中间依旧是A的东西。自嵌入描述了嵌套：α是进入另一层嵌套时生成的；β是结束这一层嵌套时生成的。嵌套最著名的例子是在算术表达式中括号的使用：

```
arith_expression_s  →  ··· | simple_expression
simple_expression   →  number | '(' arith_expression ')'
```

一个非终结符可以同时时左递归的和右递归的；它就是自嵌入的。**A--->Ab|cA|d**就是一个例子。

如果一个语法中没有非终结符时递归的，则每一个生成步骤使用一个非终结符，因为这个非终结符绝不会再次出现在这个段中。所以生成过程不可能无限持续，结果就产生了一个有限的语言。递归是语法生命周期所必须的。

2.3.2.2 一些例子

在现实世界中，很多事物是根据其他事物来定义的。上下文无关语法是一种非常简洁的方式来制定这种相互关系。一个最浅显的例子就是一本书的组成，如图Fig 2.10所示。

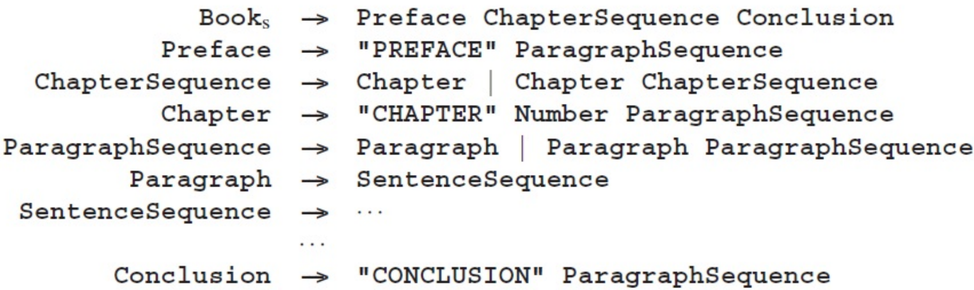


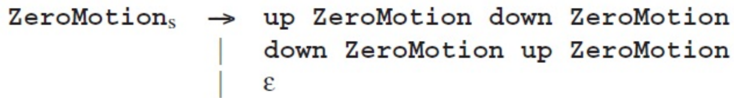
Fig. 2.10. A simple (and incomplete!) grammar of a book

当然这是关于一本书的上下文无关描述，所以人们也可以期待其生成很多冠冕堂皇的废话，就像

```
PREFACE
qwertyuiop
CHAPTER V
asdfghjkl
zxcvbnm,.
CHAPTER II
qazwsxedcrfvtgb
yhnujmikolp
CONCLUSION
All cats say blert when walking through walls.
```

不过至少可以保证结果能得到正确的结构。文档编制和文本标记系统，如SGML、HTML、XML使用这种方式来表达和控制文档的基本结构。

一个比较简短但相对重要的例子是，使所有电梯回到原始出发点的运动的语言（被限制在第五大道的Manhattan龟也会有一样的运动轨迹）



（我们假设电梯井使无限长的；第五大道也一样）。

如果我们忽略足够多的细节，我们也能认识到自然语言的句子中底层的上下文无关结构，例如英语：

```
     Sentence_S  →  Subject Verb Object
        Subject  →  NounPhrase
         Object  →  NounPhrase
     NounPhrase  →  the QualifiedNoun
  QualifiedNoun  →  Noun | Adjective QualifiedNoun
           Noun  →  castle | caterpillar | cats
      Adjective  →  well-read | white | wistful | ···
           Verb  →  admires | bark | criticize | ···
```

英语中生成像这样的句子：

the well-read cats criticize the wistful caterpillar

然而，因为没有上下文被纳入，它也会生成不不正确的形式

the cats admires the white well-read castle

为了保持上下文我们需要使用短语结构语法（为了更简单的语言）：

```
     Sentence_S  →  Noun Number Verb
         Number  →  Singular | Plural
  Noun Singular  →  castle Singular | caterpillar Singular | ···
  Singular Verb  →  Singular admires | ···
       Singular  →  ε
    Noun Plural  →  cats Plural | ···
    Plural Verb  →  Plural bark | Plural criticize | ···
         Plural  →  ε
```

标记Singular和Plural控制实际英文单词的生成。仍然，这个语法允许猫吠叫...。一种更好的控制上下文的方式，详见15章的各小节，尤其是Van Wijngaarde语法（15.2节）以及属性和词缀语法（15.3节）。

大部分的CF语法的列子来源于编程语言。这些语言（也就是程序）中的句子都必须自动处理（也就是，通过编译器），而且很快（大约1958年）人们就承认如果语言有一个良好定义的正式语法将会变得更容易。现今使用的所有编程语言的句法都是通过正式语言定义的。

一些作者（例如Chomsky）和一些解析算法，要求CF语法是单调的。CF规则为非单调的唯一方式是右侧为空。这一规则被称为ε法则，而不包含这类法则的语法被称为免ε。

免ε并不是一种限制，而只是一种妨碍。几乎任何一种CF语法都可以变成免ε的，通过ε法则的系统性替换；唯一的例外是一个语法的开头符号已经生成了ε。转换过程在4.2.3.1节有详细介绍，不过这里同时也介绍了其他很多语法的转换，而且不幸的是通常会破坏语法的结构。这个问题将在2.5节进一步讨论。

2.3.2.3 注释样式

编程语言中的CF语法有几种不同的注释样式，每个都有无尽的变形；不过它们的功能都是一样的。这里我们给出了两种主要形式。第一个是Backus-Naur形式(BNF)，最初用来定义ALGOL 60的。示例如下：

```
<name>::=         tom | dick | harry
<sentence>s::=    <name> | <list> and <name>
<list>::=         <name>, <list> | <name>
```

这个形式的主要属性是通过使用尖括号（<>）将非终结符括起来，"::="来表示“可能生成”。在一些变形中，由分号结尾。

第二种样式是CF van Wijngaarden语法的。示例如下：

```
name:         tom symbol; dick symbol; harry symbol.
sentences:    name; list, and symbol, name.
list:         name, comma symbol, list; name.
```

结尾符号用...符号；它们的表示是硬件依赖的而且不在语法中定义。规则都是正确终止的（以句点结束）。标点符号总是或多或少的按照传统方式使用；例如，逗号联结比分号要紧密。标点符号如下：

:	“is defined as a(n)”
;	“, or as a(n)”
,	“followed by a(n)”
.	“, and as nothing else.”

上述语法的第二条规则应该读作：“一个sentence被定义为一个name或一个list，其后跟着一个and-symbol然后跟着一个name，除此没有别的了。”虽然这种表达方式仅只在适用于两级Van Wijngaarden语法时能得到最大展现，它依旧有自己的优点：它非常正式且可读性很强。

2.3.2.4 CF语法扩展

CF语法通常是既比较紧凑又可读性较高的，通过为频繁使用的结构引入特殊的短项。如果我们回到图 2.10中的书本结构语法，将会看到类似下面这种规则频繁出现：

SomethingSequence ---> Something | Something SomethingSequence

在上下文无关语法的扩展中，我们可以用**Something**$^{+}$来表示“one or more **Something**”，而且我们不需要专门给出**Something**$^{+}$的规则；于是有了下面的隐式显示：

Something+ ---> Something | Something Something+

同样，我们可以用**Something***来表示“零个或多个Somethings”，以及**Something**$^{?}$来表示“零个或一个**Something**”（也就是说，可选择的**Something**）。这些例子中，符号$^{+}$,*和$^{?}$要与前面的符号一起起作用。它们的范围可以扩展，通过括号：**(Something ;)**$^{?}$来表示“选择一个Something-followed-by-a-;”。这些措施非常有用并且让书本结构的语法可以更高效的写出来（图Fig 2.11）。有一些样式甚至允许**Something**$^{+4}$这样的结构，用以表示“一个或不超过4个的Somethings”，或者是**Something**$^{+}$**,**来表示“一个或多个通过逗号分隔的Somethings”；这似乎可以作为把一件好事做过头的例子。这种表示法被称为BNF扩展（EBNF）。

```
     Books  →  Preface Chapter+ Conclusion
   Preface  →  "PREFACE" Paragraph+
   Chapter  →  "CHAPTER" Number Paragraph+
 Paragraph  →  Sentence+
  Sentence  →  ...
               ...
Conclusion  →  "CONCLUSION" Paragraph+
```

Fig. 2.11. A grammar of a book in EBFN notation

EBNF语法扩展不会增加其表现力：所有隐式规则都可以变成显式，结果得到一个BNF的正常CF语法注释。它们的力量在于其用户友好性。X^*的星号表示“一连串的零个或多个*X*”，其被称为Kleene星号。如果X是一个集合，那X^*应该表示“一连串的零个或多个元素X”；这与我们在2.1.3.3节中提到的Σ^*中的星号一样。涉及到重复运算符*,$^+$或者$^?$以及分隔运算符 **(** 和 **)** 的运算形式被称为正则表达式。右侧拥有正则表达式的EBNF，就是其偶尔被称为右侧正则语法*RRP*语法（regular right part grammar）的原因，这个称呼比“上下文无关语法扩展”要更具表现性，不过也有人说这个名称有点像绕口令。

关于RRP语法的结构意义有两个不同的思想流派。一个流派主张规则应该是这样的：

Book ---> Preface Chapter+ Conclusion

上述应该是下面的缩写：

Book ---> Preface α Conclusion α ---> Chapter | Chapter α

这是“（右）递归”的解释。它的优点是易于解释而且转换到“常规”CF语法很简单。缺点是转化过程包含了隐式规则（用α表示）而且，例如一本有四章的书的不平衡生成树不对应于我们关于这本书的结构的本来想法；见图Fig 2.12。

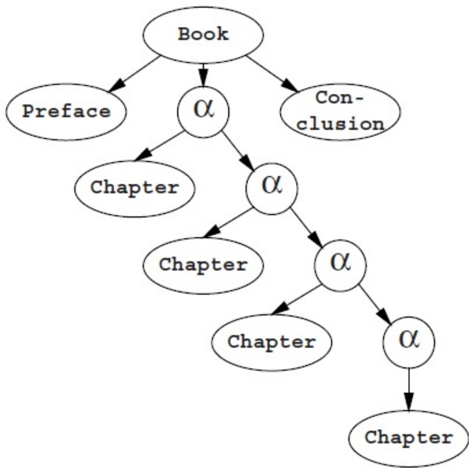


Fig. 2.12. Production tree for the (right)recursive interpretation

第二种流派主张

Book ---> Preface Chapter+ Conclusion

上述是下面的缩写：

```
Book  →  Preface Chapter Conclusion
       |  Preface Chapter Chapter Conclusion
       |  Preface Chapter Chapter Chapter Conclusion
       |  ...
       ...
```

这是“迭代”的解释。它的优点是能产生一个优美的生成树（图Fig 2.13），但是缺点是它涉及到了无限多的生成规则而且生成树的节点有着变化的fan-out。

因为迭代解释的实施是不会妄自菲薄的，最具实用性的解析生成器在一些时候使用递归解释，然而大多数研究都是关于迭代解释的。

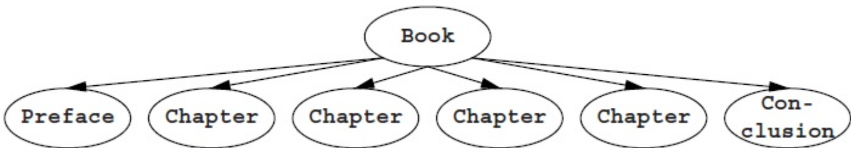


Fig. 2.13. Production tree for the iterative interpretation

2.3.3 3型语法

CF语法的基本属性是它描述的是嵌套：一个对象可能在不同地方包含其他对象，这些对象又可能包含其他对象...等等。在生成过程中我们已经生成了对象中的一个，而右侧依旧“记得”接下来该生成什么：在英语语法中，在升入非终结符**Subject**之后生成类似**wistful cat**的东西，右侧的**Subject Verb Object**依然知道后面必须是一个**Verb**。当我们正在准备**Subject**时，**Verb**和**Object**已经在句子形式的右侧排队等待了，例如：

the wistful QualifiedNoun Verb Object

右侧是：

up ZeroMotion down ZeroMotion

已经执行了一个**up**和一个任意的复杂的**ZeroMotion**后，右侧依旧知道接下来是一个**down**。

对3型的限制不允许回溯以前的事情：一个右侧可能只包含一个非终结符并且其必须在结尾处。这意味着只有两种规则：[1]

- 一个非终结符生成零个或多个终结符
- 一个非终结符生成零个或多个其后接着一个非终结符的终结符

3型语法的原始Chomsky定义对规则进行了限制：

- 一个非终结符生成一个终结符。
- 一个非终结符生成一个其后接着一个非终结符的终结符。

我们的定义是等效的并且更加方便，虽然转换为Chomsky3型并不是完全没有意义的。

3型语法也被称为正则语法（RE语法）或有限状态语法（FS语法）。以上版本的更精确定义称为右正则，因为规则中唯一的非终结符出现在右侧的右结尾处。这让其有别于左正则语法，其受到以下限制

- 一个非终结符生成零个或多个终结符
- 一个非终结符生成一个其后接着零个或多个终结符的非终结符

其规则中唯一的非终结符出现在右侧的左结尾处。左正则语法相比于右正则语法不那么直观，且不常出现，以及更难处理，但它们确实偶尔会出现（例子见5.1.1节），并且也需要加以考虑。将会在5.6节进行讨论。

由于右正则语法的普遍性高于左正则语法，“正则语法”一词通常是指“右正则语法”，左正则就需要明确指明了。本书中我们也遵照此约定。

比照右正则和右递归的定义是一件有趣的事情。非终结符A是右递归的，如果它能生成一个右侧结尾是A的句子形式；或A是右正则的，如果它能生成一个包含A的句子形式，且A在其右侧结尾处。

在对上下文无关语法的类比中，其被称为在此之后它们不能做到，正则语法应该被称为“非嵌套语法”。

因为正则语法经常被用来描述字符级的文本结构，所以正则语法的终结符号是一个单一字符是很常见的。所以我们应该用**t**代替**Tom**，**d**代替**Dick**，**h**代替**Harry**以及**&**代替**and**。图Fig 2.14（a）以这种形式展示了**t,d&h**语言的右正则语法，Fig 2.14（b）展示的是左正则语法。

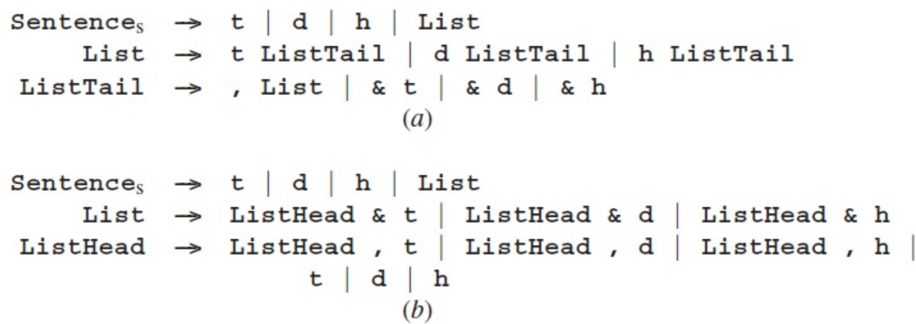


Fig. 2.14. Type 3 grammars for the t, d & h language

3型语法（右正则）的一个句子的生成树退化为非终结符的“生成链”，该链在左侧放置了一列终结符。图Fig 2.15展示了一个例子。相似的生成链由左正则语法形成，其终结符放在左侧。

图Fig 2.14中语法展现出来的致命重复对正则语法来说很正常，然后发明了很多符号处理设备来减弱这个问题。最常用的是使用一组方括号来指出“一组字符集之外的一个”：**[tdh]**是**t|d|h**的缩写：

```
Ss  ->  [tdh] | L
 L  ->  [tdh] T
 T  ->  , L | & [tdh]
```

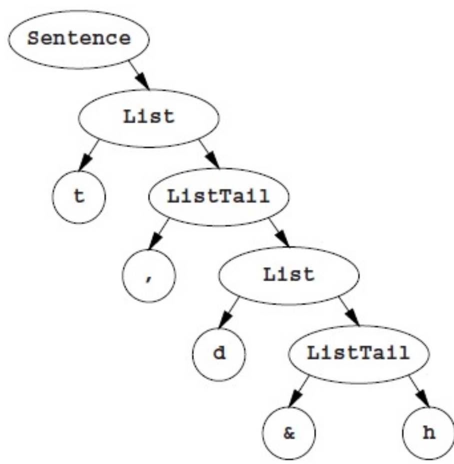


Fig. 2.15. Production chain for a right-regular (Type 3) grammar

这个第一次看上去可能很神秘，但实际上却更方便，而且能将语法简化至：

```
Ss  ->  [tdh] | L
 L  ->  |[tdh] , L | [tdh] & [tdh]
```

第二种方式是引入宏，语法片段的代名词，其在被使用前被恰当的在语法中进行替代：

```
Name  ->  t | d | h
  Ss  ->  $Name | L
   L  ->  $Name , L | $Name & $Name
```

正则语法的流行解析生成器*lex*（Lesk and Schmidt [360]）都表现了这些便利性。

如果我们坚持3型的Chomsky定义，那我们的语法将被不会小于：

```
Ss  →  t | d | h | t M | d M | h M
M   →  , N | & P
N   →  t M | d M | h M
P   →  t | d | h
```

这个形式比较容易处理，但是比起*les*版本缺少用户友好性。我们在这里看正式的语言学家（formal-linguist）是否感兴趣且是否对我们有帮助通过微小的方式，计算机科学家重视语法（**$Name**, etc.）之下的概念被清楚表达的形式，虽然有需要额外处理的代价。

关于正则语法有两个有趣的观点，我们来看一下。第一个，当我们使用正则语法来生成句子，句子形式将会只包含一个非终结符并且它将总是在结尾处；如下图（使用图Fig 2.14中的语法）：

```
Sentences
List
t ListTail
t , List
t , d ListTail
t , d & h
```

第二个观点是，所有的正则语法都可以缩减体积通过使用2.3.2节中介绍的正则表达式操作符*，$^+$和$^?$，来分别表达“零个或多个”，“一个或多个”以及“选择一个”。使用这些运算符和**(**以及**)**来进行分组，我们就可以将语法简化为：

S_S ---> (([tdh],)* [tdh]&)$^?$ [tdh]

这里用括号来标定操作符*和$^?$的操作数。所有3型语法都存在正则表达式。注意，*和$^+$与它们前面的字符一起生效。为了将它们和普通的乘法和加法运算区分开，它们通常是作为上角标打印的，但是在计算机输入中它们和另外两个符号是在同一行上的，所以必须用其他方式来进行区分。

> 1. 有一个自然的中间类型，2.5型，其中只允许一个右侧有一个单一的非终结符，不过不需要一定在结尾处。这就给了我们所谓的线性语法。↩

2.3.4 4型语法

我们现在应该谈到的在生成规则中的最后一条限制是：右侧不允许出现非终结符。这从机制上移除了所有的生成力，除了被选中的可选项。起始符号有一列（有限的）可选项，我们所允许的；这个在名称中有所展现，有限选项语法（FC语法）。

对语言**t,d&h**是没有FC语法的；然而，如果我们愿意将其限制在数量有限（例如不超过一百长度）的名字列表中，那就可以有一个FC语法了，因为可以枚举所有的组合。对于这显然在限制内的三个名字，我们就有了：

S_S ---> [tdh] | [tdh] & [tdh] | [tdh] , [tdh] & [tdh]

以上总计 3+3×3+3×3×3 = 39 条生成规则。

FC语法不是官方Chomsky层次结构的一部分，因为这不是Chomsky定义的。不过它们仍然非常有用，而且常在一些处理或推理中被要求使用。可以用FC语法来描述编程语言中的保留字段集（关键字）。虽然不是很多语法完全是FC，不过很多语法的一些规则是有限选项的。例如，我们第一个语法的第一条规则（图Fig 2.3）就是FC。另一个FC规则的例子是2.3.3节中介绍的宏。我们不需要宏机制，如果我们这样改改：

zero or more terminals

上述正则语法定义改为：

zero or more terminals or FC non-terminals

最后，FC非终结符只会引入有限的终结符。

2.3.5 结论

图Fig 2.16的表格总结了在生成一个字符串时会出现的最复杂的数据结构，在相关语法类型中使用到的。参见图Fig 3.15，在解析中获得的相应的数据类型。

Chomsky type	Grammar type	Most complicated data structure	Example figure
0 / 1	PS / CS	production dag	2.8
2	CF	production tree	2.9
3	FS	production list	2.15
4	FC	production element	—

Fig. 2.16. The most complicated production data structure for the Chomsky grammar types

2.4 用语法生成句子

2.4.1 短语结构案例

直到现在，我们用语法只生成了一个句子，一个特定的时尚，不过语法的目的是生成所有句子。幸运的是还有系统可以做到这个。我们用$a^nb^nc^n$语法来做例子。我们从起始符号开始，然后用系统尝试所有可能的替换来生成所有的句子形式；我们只需要等着看哪些句子在什么时候演变成句子。手动完成10个句子试试。如果我们不小心，很容易就会生成像**aSQ, aaSQQ, aaaSQQQ,...**这样的形式，而且我们将永远也看不到一个完整的句子。原因是我们太过专注于一个单一的句子形式：我们应该给所有的句子同样的时间。这可以通过下面的算法完成，其保持句子形式的一个队列（也就是一个列表，表中是我们要加到结尾和从开头删除的元素）。

从起始符号开始，作为队列中唯一的句子形式。现在继续以下操作：

- 考虑队列中的第一个句子形式。
- 从左到右扫描它，寻找符合左侧生成规则的字符串。
- 发现的每一个这样的字符串，复制足够的句子形式，替换每一个符合左侧生成规则的字符串，通过规则中不同的选项，然后把它们全部添加到队列末尾。
- 如果原始句子形式不包含任何非终结符，把它作为语言中的一个句子写下来。
- 扔掉原始的句子形式；它已经被处理完成了。

如果没有匹配的规则，并且句子形式不是一个完成的句子，那这就是一条死胡同；它们会被上述过程自动删除，不留任何痕迹。

因为上述过程枚举了PS语言中所有的字符串，PS语言也被称为递归可枚举集，这里的“递归”是指“通过一个可行的递归算法”。

图Fig 2.7中$a^nb^nc^n$语言的处理过程的开始几步，展示在图Fig 2.17中。队列向右运行，其第一项在左侧：

Step	Queue					Result
1	S					
2	abc	aSQ				abc
3	aSQ					
4	aabcQ	aaSQQ				
5	aaSQQ	aabQc				
6	aabQc	aaabcQQ	aaaSQQQ			
7	aaabcQQ	aaaSQQQ	aabbcc			
8	aaaSQQQ	aabbcc	aaabQcQ			
9	aabbcc	aaabQcQ	aaaabcQQQ	aaaaSQQQQ		aabbcc
10	aaabQcQ	aaaabcQQQ	aaaaSQQQQ			
11	aaaabcQQQ	aaaaSQQQQ	aaabbccQ	aaabQQc		
...	...					

Fig. 2.17. The first couple of steps in producing for $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$

可以看到，并不是每次我们打开曲柄都可以得到一个句子；实际上，在这种情况下真正的句子十分稀少。当然是因为在这一过程中，发展出了很多的侧边线，而这都需要同等的重视。不过我们可以肯定，每一个可以生成的句子最终都会生成：我们不会放过任何可能。这种方式被称为广度优先生成；而计算机做的比人要好。

很容易认为对我们在顶层句子形式中发现的所有左侧内容全部替换是不必须的。为什么不只替换第一个，然后等下一个句子形式出现，在替换下一个？然而这是错误的，因为替换第一个有可能会导致第二次替换时上下文混乱。一个简单的例子就是下面的语法：

$$
\begin{aligned}
S_s &\rightarrow AC \\
A &\rightarrow b \\
AC &\rightarrow ac
\end{aligned}
$$

第一次替换**A--->b**将会走到死胡同里，那语法将什么也不会生成。做两个可能的替换也将导致同样的死路，但是却依旧会有第二个句子形式，**ac**。也有一个语法例子，其中队列在一段时间（很短）后将会变空。

如果语法是上下文无关的（或正则），那就不会有所谓的上下文被破坏了，那么替换第一个（或只替换）匹配的就是很安全的了。

这里有两点要说明。第一，经过我们努力之后得到的句子并不一定就会是我们想要的那个：很可能每一个新的句型又包含着一个非终结符。我们应该通过检查语法事先知道，不过可以证明对PS语法来说是不可能做到的。正式的语言学家（formal-linguist）说“一个PS语法是否生成空集是不可判定的”，这意味着，没有一个算法能明确得出每一个PS语法最终是否能生成一个句子。不过这不意味着我们不能对于给定的算法证明其无法生成句子，如果该语法是这样的情况。这意味着证明方法不适用于所有的语法：在一定时间内能完成那我们就可以有一个程序，而如果是不能完成的那时间就无法估测了。实际上，上述生成过程就是一个确切告诉我们答案是“是\否”的算法（虽然我们可以有一个告诉我们“是\不知道”的算法，在有限时间内）。虽然由于语言的一些深层属性导致我们不能总是确切得到我们想要的答案，但这并不会妨碍我们获取各种信息以更加了解语法。我们应该看到这是一种反复出现的现象。计算机科学家知道形式语言学的绝境，但他们并不气馁。

第二点是，当我们确实从上述生成过程中得到句子的时候，它们却可能是通过不可探寻的顺序生成的。对于非单调语法，句子形式可能短暂增加然后急剧缩小，甚至可能变成空字符串。形式语言学证明，不会有一个适用所有PS语法的算法来生成长度增加（实际上是“非递减”）的句子。换句话说，PS语法的解析问题是不可解决的。（虽然术语是可以互换的，但似乎使用“不可判定”来描述“是\否”问题以及“不可解决”来描述“解析问题”要更合理。）

2.4.2 CS案例

上述的语言生成过程也适用于CS语法，除了有关确定性的那部分。因为在进行中的句子形式绝对不会收缩，所以句子的长度是以单调递增顺序生成的。这意味着如果空字符串不是第一个字符串，那它将再也不会出现了，而且CS语法不会生成ε。此外，如果我们想知道一个给定字符串*w*是否在语言中，我们可以等着看它是否会出现，如果出现那么答案就是“是”，而如果我们看到生成了长度更长的字符串，那答案就是“否”。

由于CS语言的字符串可以被一个可能的递归算法识别，所以CS语言也被称为递归集。

2.4.3 CF案例

当我们通过CF语法生成句子时，很多事情都简单很多。虽然我们的语法可能永远不会生成句子依旧有可能发生，不过我们现在可以事先进行检测，如下。首先，扫描语法以找出所有的非终结符，其对应右侧拥有终结符或为空的。这些非终结符确保能生成东西。现在再次扫描找到对应右侧只有唯一终结符的非终结符，这些非终结符确保能生成东西。这将给我们新的保证能生成东西的非终结符。重复这个过程直到不在有新的非终结符产生。如果按这种方式始终找不到起始符号，那这个语法将不会生成什么了。

此外我们已经看到如果语法是CF形式的，我们就能每次重写最左侧的非终结符（直到我们重写了所有的可选项）。当然我们也可以一直重写最右侧的非终结符。这种方法类似但也有不同。通过下面的语法：

0.	N	→	t | d | h
1.	S_S	→	N | L & N
2.	L	→	N , L | N

让我们跟随句子形式历险，而这将最终导致**d,h&h**。虽然这将经历几次生成队列，但我们在这里只描述对它做了哪些改变。图Fig 2.18展示了句子形式最左侧或最右侧的替换，根据涉及到的规则和备选项；例如（1b）表示规则1备选项b，第二个备选项。

	`S`		`S`
1b		1b	
	`L&N`		`L&N`
2a		0c	
	`N,L&N`		`L&h`
0b		2a	
	`d,L&N`		`N,L&h`
2b		2b	
	`d,N&N`		`N,N&h`
0c		0c	
	`d,h&N`		`N,h&h`
0c		0b	
	`d,h&h`		`d,h&h`

Fig. 2.18. Sentential forms leading to `d,h&h`, with leftmost and rightmost substitution

生成规则使用的序列与我们所期望的不同。 Of course in grand total the same rules and alternatives are applied, but the sequences are neither equal nor each other's mirror image, nor is there any other obvious relationship. Both sequences define the same production tree (Figure 2.19(a)), but if we number the non-terminals in it in the order they were rewritten, we get different numberings, as shown in (b) and (c).

The sequence of production rules used in leftmost rewriting is called the leftmost derivation of a sentence. We do not have to indicate at what position a rule must be applied, nor do we need to give its rule number. Just the alternative is sufficient; the position and the non-terminal are implicit. A rightmost derivation is defined in a similar way.

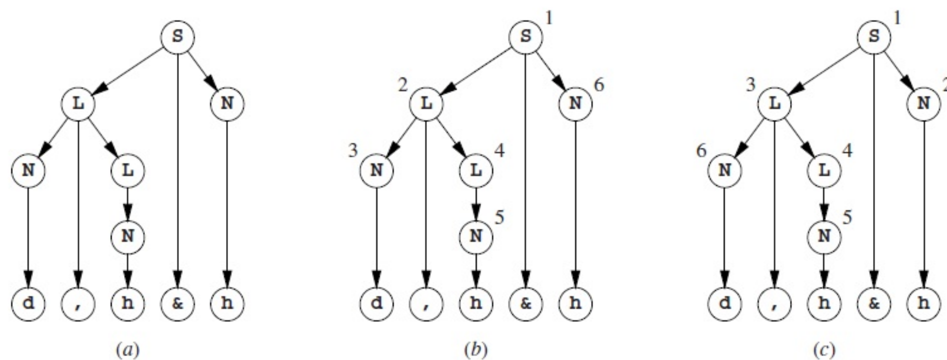


Fig. 2.19. Production tree (*a*) with leftmost (*b*) and rightmost (*c*) derivation order

A leftmost production step can be indicated by using an arrow marked with a small l: N,L&N l--->d,L&N, and the leftmost production sequence

S l---> L&N l---> N,L&N l---> d,L&N l---> d,N&N l---> d,h&N l---> d,h&h

can be abbreviated to S l--*->d,h&h. Likewise, the rightmost production sequence

S r---> L&N r---> L&h r---> N,L&h r---> N,N&h r---> N,h&h r---> d,h&h

can be abbreviated to S r--->*d,h&h. The fact that S produces d,h&h in any way is written as S*--->d,h&h.

The task of parsing is to reconstruct the derivation tree (or graph) for a given input string. Some of the most efficient parsing techniques can be understood more easily if viewed as attempts to reconstruct a left- or rightmost derivation process of the input string; the derivation tree then follows automatically. This is why the notion “[left|right]-most derivation” occurs frequently in this book (note the FC grammar used here).

2.5 收敛或不收敛

在前面的段落中，关于是否一个规则的右侧要比其左侧简短，有时我们很明确但有时我们又是含糊不清的。0型规则应该说肯定是收缩的类型，而单调型规则肯定不是，2型和3型只有在生成空集（ε）时才是收缩的；这些都是肯定的。

原始Chomsky层次结构（Chomsky[385]）在这个问题上非常坚定：只有0型规则才能使句子形式收缩。1型、2型和3型规则都是单调的。此外1型规则必须使上下文相关类型，这意味着左侧非终结符的只有一个是被允许替换的（且不由ε替换）。这带来了一个合适的层次，使得每一层都是其父集的一个合适子集，以及使除了0型语法外的所有派生图实际上都是派生树。

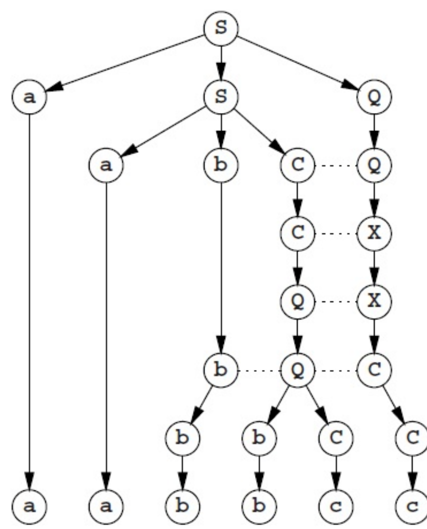
作为一个例子，考虑一下图Fig 2.7中语言$\mathbf{a^n b^n c^n}$的语法：

1.	S_s	→	abc | aSQ
2.	bQc	→	bbcc
3.	cQ	→	Qc

它是单调的，但不是严格意义上的上下文相关。可以通过展开那烦人的规则3和为**c**引入一个非终结符来使其成为CS：

1.	S_s	→	abC | aSQ
2.	bQC	→	bbCC
3a.	CQ	→	CX
3b.	CX	→	QX
3c.	QX	→	QC
4.	C	→	c

现在图Fig 2.8的生成图变成了一个生成树：



还有另一个理由回避ε规则：它让定理和解析器都变得更复杂，有时尤其复杂；例如9.5.4节。因此就出现了一个问题，我们到底是为什么要纠结ε规则呢；答案很简单因为这对语法作者和使用者来说十分方便。

如果有一个语言是使用ε规则的CF语法来描述的，而我们想要使用一个不含ε规则的语法来进行描述时，那这个语法将会是非常复杂的。假设我们有一个系统，可以输入比特信息，比如“Amsterdam is the capital of the Netherlands”，“Truffles are expensive”，那就会被问道一个问题了。在一个非常浅显的水平上，我们可以将其输入定义为：

input$_S$: zero-or-more-bits-of-info question

或者，以一种扩展的记法

input$_S$: bit-of-info* question

因为**zero-or-more-bits-of-info**将会生成空字符串，在其他字符串之间，至少在其语法中使用的规则之一是ε规则；在扩展记法中的 * 已经意味着ε规则了。 在使用者的角度来看，上述输入的定义很好的解释了这个问题，并且就是我们想要的。

任何试图为这个输入写ε规则，最终都会定义一个概念，包含后来的bits-of-info和问题一起（因为这个问题是唯一的非空部分，所以它必须出现在所有有关的规则中）。但是这个定义根本不是我们想要的，而且它是一个半成品：

```
                     inputs: question-preceded-by-info
question-preceded-by-info: question
                           | bit-of-info
                               question-preceded-by-info
```

随着语法变得越来越复杂，其是ε无关的要求就变得越来越令人讨厌：语法在和我们作对，而不是在为我们工作。

从理论角度来看这不成问题：任何CF语法都能被一个ε无关语法描述，并且ε规则在也不被需要。更妙的是任何带有ε规则的语法都能被转化为ε无关的语法，作为同一种语言。在以上示例中我们看到了这种转变，而算法详细将在4.2.3.1节讲述。但是我们付出的代价是，对任何语法的转换：这不在是我们的语法，并且它极少的反应原始结构。

底线是研究人员发现ε规则是一个有用的工具，并且除了通常的Chomsky层次结构外，是否存在非单调语法的层次结构，我们拭目以待。一个更大的扩展：2型和3型语法不需要是单调的（因为如果有需要，它们总是可以变成这样）；并且收敛的上下文相关语法总是等同于无限制的0型语法；而蕴含ε规则的单调语法总是等同于0型语法。系暗转我们可以把这两个层次画在一张图里面；见图Fig 2.20。将不同作用的语法类型用线条分隔。作用相同但理论上不同的语法用空格分隔。可以看到，如果我们坚持非单调性，那0型和1型的区别就消失了。

		Chomsky (monotonic) hierarchy		non-monotonic hierarchy
global production	Type 0	unrestricted phrase structure grammars	monotonic grammars with ε-rules	unrestricted phrase structure grammars
	Type 1	context-sensitive grammars	monotonic grammars, no ε-rules	context-sensitive grammars with non-monotonic rules
local production	Type 2	context-free ε-free grammars		context-free grammars
	Type 3	regular (ε-free) grammars		regular grammars, regular expressions
no production	Type 4	finite-choice		

Fig. 2.20. Summary of grammar hierarchies

如果1型到3型语法本身包含空字符串，那就出现一个特殊的情况。这不能被纳入单调层次结构的语法中，因为其起始符号长度已经为1且没有单调规则能让它收敛。所以空字符串应该被重视作为语法的一个特殊属性。这样的问题不会出现在非单调层次结构中。

许多解析方法原则上只为ε无关语法工作：如果一个东西什么都不能产生，那你可能不太容易发现它是否在那。通常解析方法可以修改来控制ε规则，但这总是会增加方法的复杂度。这么说可能也是公平的，这本书将薄30%，如果ε规则不存在的话——不过，语法就要损失不止30%的价值了！

2.6 生成空语言的语法

在印度当0被作为数字被数学家引入后的大约1500年后，这一概念依旧没有被计算机科学好好接受。许多编程语言不支持0字段的记录，0元素的数组，或0变量的变量定义；在一些编程语言中，调用0个参数协程的语法不同于调用一个或多个参数的协程的语法；许多编译器也无法编译定义0个名称的模块；这些例子还可以很轻松的扩展下去。更具体的说，我们不知道有什么解析器生成器可以生成一个空语言的解析器，这个空语言有0个字符串。

所有这一切都将我们引向一个问题，空语言的语法会是什么样的？首先注意，空语言不同于只包含空字符串的语言，空字符串只是包含0个字符。这种语言很容易由**S_S--->ε**生成，并且被普通*lex-yacc*流正确处理。注意，这个语法没有终结符，这意味着2.2节的VT是空集。

对一个语法来说不生成任何东西，那生成过程将不被允许终止。这就有了一个办法来获得这一一种语法：**S_S--->S**。然而这很令人不齿，有两个原因。从算法角度来说，生成过程只是在循环，而没有获得任何关于语言的空属性的信息；而且符号**S**的使用是任意的。

另一个方法就是迫使生成过程被卡住，通过让语法中没有任何生成规则。那么2.2节的*R*就也会是空的了，然后语法的形式就是({S}, {}, S, {})。这还不是很理想，因为我们由一个没有定义规则的非终结符；并且符号**S**还是任意的。

更好的方式是永远不要让生成过程开始：没有起始符号。这是可行的，通过在语法定义中允许出现一组起始符号而不是一个单一的起始符号。这么做还有其他很好的理由。举个例子，一个大型编程语言的语法，该语言的模块规格、模块定义等有着多重“根”。虽然在顶层这些是不同的，但它们在共同语法上有着大量的段（segment）。如果我们将一个CF语法定义扩展来使用一组起始符号，空语言的语法将获得优雅和令人满意的形式({}, {}, {}, {})。

关于0和空：考虑一下左侧是空的语法规则，可能是有帮助的。这种规则的右侧的最终生成物可能出现在输入的任何地方，因此模拟噪声和其他每天但外部的事件。

我们全神贯注于空字符串、空集、空语言等不是轻浮的，因为这是众所周知的，系统处理空实例的轻松是其洁净度和稳健性的一个衡量。

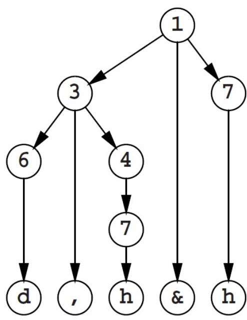
2.7 CF和FS语法的限制

当使用CF语法工作一段时间后，你就会渐渐感觉到似任何东西都能用一个CF来进行描述。然而，CF语法能描述的却有着严重的限制，按照著名的*uvwxy*理论的说法，下面将进行解释。

2.7.1 *uvwxy*理论

当我们从CF语法中获得了一个句子后，我们应该仔细看句中的每一个终结符，然后问自己：它是怎么在这的？然后看一下生成树，我们看到它被生成作为右侧规则m的第n位成员。这个规则的左侧，符号的父节点，再次生成规则Q的第p位成员，等等，直到我们到达起始符号。在某种意义上，我们可以通过这种方式追踪符号的轨迹。如果一个符号的轨迹的所有规则/成员对都是不同的，我们称这个符号是原始的，如果一个句子中的所有符号都是原始的，我们称这个句子是“原始”的。

例如，下面生成树中第一个**h**：



由下面语法生成的

1. S_S → L & N
2. S_S → N
3. L → N , L
4. L → N
5. N → t
6. N → d
7. N → h

h的谱系是，来自**7**，**1**，来自**4**，**1**，来自**3**，**3**，来自**1**，**1**.这里，第一个数字代表规则数，第二个数字代表这个规则中的成员数。因为所有的规则/成员对都是不相同的，所以**h**是原始的。

现在对于一个给定的符号，只有有限的方式来让其是原始的。这很容易，如下所示。一个原始符号的谱系中的所有规则/成员对必须是不相同的，所以其谱系的长度一定不会比语法中所有不同规则/成员对的总长度还长。There are only so many of these, which yields only a finite number of combinations of rule/member pairs of this length or shorter. 理论中，一个符号的原始谱系的数量可能是非常巨大的，但实际中确实非常小的：如果有超过10种方法来生成一个给定的符号，从语法中通过原始谱系，那这个语法将会是非常错综复杂的！

这对原始句子就有了严格的限制。如果原始句子中一个符号出现两次，这两个的谱系必须不相同：如果谱系相同，那它们应该描述的是用一个位置的同一个符号。这意味着原始长度有着最大长度：所有符号的原始谱系的长度总和。对一个编程语言语法的平均的长度，在数以千计的符号的长度顺序中，大致相当于语法的长度。所以，因为有着最长的原始句子，那么就只能有着有限数量的原始句子，然后我们就得到了一个令人惊讶的结论就是任何CF语法只生成大小有限的原始句子核心，以及（也许）无限数量的非原始句子！

“非原始”句子是什么样的？这就是我们开始介绍*uvwxy*定理的地方。一个非原始句子具有这一的属性，在谱系中包含至少一个重复出现的符号。假设那个符号是*q*，重复的规则是*A*。那我们就可以画一幅累死图Fig2.21的图，*w*是由*A*的最新应用生成的部分，*vwx*是*A*的其他应用生成的部分，*uvwxy*就是这个非原始句子。现在我们立即就可以找到另一个非原始句子，通过删除以*A*为首的小三角，然后用以*A*为首的大三角副本替换；见图Fig 2.22。

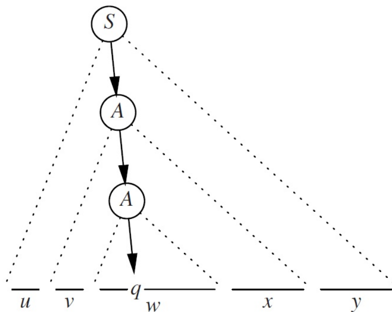


Fig. 2.21. An unoriginal sentence: *uvwxy*

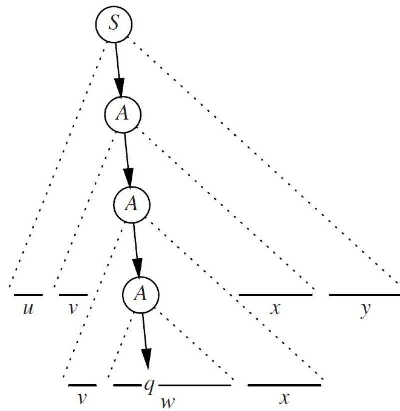


Fig. 2.22. Another unoriginal sentence, uv^2wx^2y

这个新的树生成了句子*uvvwxxy*，并且以这种方式，可以很容易看到我们能构建句子uv^nwx^ny的完整家族对于所有n ≥ 0的。这种方式显示了*w*嵌套于一组*v*和*x*括号之间，在*u*和*y*无关的上下文中。

底线是当我们审查一个上下文无关语言中的越来越长的句子，原始句子慢慢用尽，我们遇到的只是句子的相近形式的家族，慢慢缩进至无限。这在*uvwxy*理论中有总结：任何由CF语法生成的句子，比最长的原始句子还要长的句子，都可以被切分称五个部分*u,v,w,x,y*，以这种方式uv^nwx^ny就是这个语法下n ≥ 0得来的一个句子。*uvwxy*理论也被称为上下文无关语言的泵引理（pumping lemma），而且由几个变种。

有两点必须在这里指出。第一点，如果一个语言持续生成越来越长的句子，而不减少嵌套句子的族系，那这个语言就不会存在一个CF语法。我们已经遇到了上下文相关语言$\mathbf{a^nb^nc^n}$，很容易看到（但不是很容易去证明！）它不会衰退成这样的嵌套的句子，当句子变得越来越长时。因此，它是没有CF语法的。这种证明的通用技术见Billington [396]。

第二点是，最长的原始句子是语法的一个属性，而不是语言的。通过为语言制造一个并发的语法，我们能增加原始句子的集合，并且可以推开我们被迫诉诸于嵌套部分的边界。如果我们让语法无限并发，那我们就能使边界变得无限，并从中获得一个短语结构语言。如何将CF语法变得无限并发，将会在15.2.1节中的两级语法中介绍。

2.7.2 *uvw*理论

*uvw*理论的简单形式应用于正则（3型）语言。我们已经看到FS语法生成的句子形式全部都仅只包含一个非终结符，在结尾出现。在一个很长的句子的生成期间，一个或多个非终结符必须出现两次或多次，因为只能有有限数量的非终结符。图Fig2.23展示了当我们一个一个列出句子形式时所看到的。

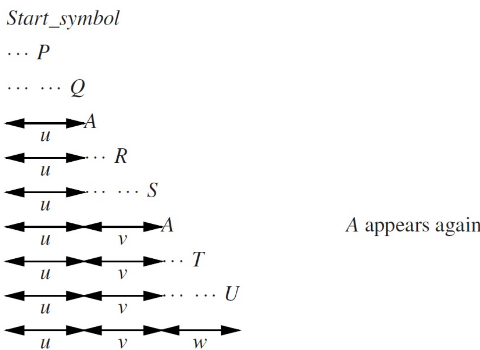


Fig. 2.23. Repeated occurrence of *A* may result in repeated occurrence of *v*

子串*v*在*A*的一次出现到下一次之中被生成，*u*是让我们能到达*A*的一个序列，*w*是让我们能终止生成过程的序列。将明确指出，从第二个*A*开始，我们可能重复了跟第一个*A*一样的路径，从而生成了*uvvw*。这将我们引导到*uvw*理论，或正则语言的泵引理（*pumping lemma for regular languages*）：正则语言的任何一个足够长的字符串都能被切分成*u*，*v*，*w*3个部分，所以对n ≥ 0的所有$uv^n w$都是这个语言的一个字符串。

2.8 作为转换图的CF和FS语法

转换图是一个有向图，其中箭头被标记为零或你生成的一个相关联的符号，如果确实有这样的符号的话，否则就什么也不标记。节点，往往没有标记的，是符号在生成中被放置的点。如果一个节点有多个箭头向外传出，你可以选择任何一个继续往下。所以图Fig 2.24中的转换图产生相同的字符串，2.3.2节的示例语法。

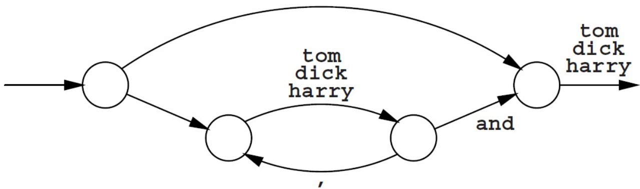


Fig. 2.24. A transition graph for the `[tdh]` language

把语法变成一组转换图是相当简单的，一个非终结符对应一个转换图，如图Fig 2.25。但它包含标记了非终结符的箭头，而“生成”一个非终结符的意义与箭头相关联并不是非常明确。假设我们在节点n_1，从一个标记了非终结符的转换（箭头）N指向节点n_2，而且我们想要这种转换。

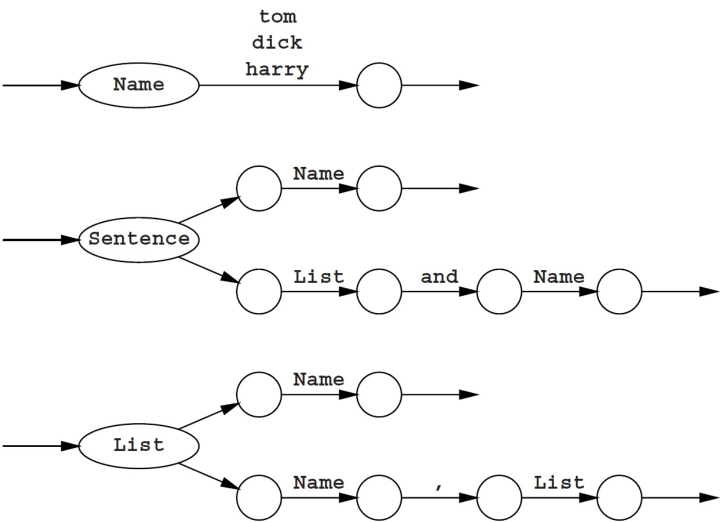


Fig. 2.25. A recursive transition network for the sample grammar on page 23

我们将节点n_2推入堆栈，而不是通过追加到输出来生成N，然后继续我们的进入N的转换图的旅程。当我们结束N的转换图是，我们从堆栈中弹出n_2然后在N_2继续向前。这就是上下文无关语法的递归传递网络释义：转换图组就是传递网，堆栈机制提供递归。

图Fig 2.26展示了图Fig 2.14中FS语法的右正则规则的转换图。这里我们漏掉了图终点处未标记的箭头和与其相关联的节点；我们本可以和图Fig 2.25中一样做，但这么做将会使堆栈机制变复杂。

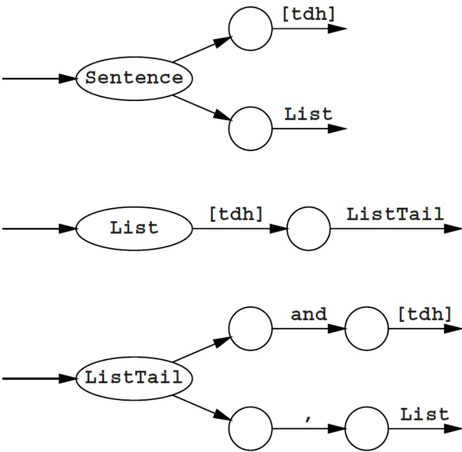


Fig. 2.26. The FS grammar of Figure 2.14(*a*) as transition graphs

我们看到只有当我们要离开一个非终结符的时候我们才需要在生成一个，所以我们不需要堆栈任何东西，并且能解释一个标记了非终结符*N*的箭头作为到*N*的转换图的跳转。所以一个正则语法对应于一个（非递归）传递网络。

如果我们将网络结尾处的每一个标记了*N*的箭头和*N*转换图的起始处相关联，那我们可以忽略掉非终结符，然后获得相关语言的一个转换图。当我们将这个短路劲应用在图Fig 2.26的传递网络，并稍微重新排列一下节点，我们就得到了图Fig 2.24的转换图。

2.9 上下文无关语法的健全

所有种类的语法都可能包含无用的规则，这些规则在任何成功的生成过程中都不能成为一个有用的角色。一个生成过程是成功的，当它一一个终结符结尾时。生成尝试可能失败，通过卡住（下一步没有可替代的）或者进入一种没有替代序列能移除掉所有的非终结符的境地的情况。0型语法被卡住的一个示例如下：

1.	S_S	→	A B
2.	S	→	B A
3.	S	→	C
4.	A B	→	x
5.	C	→	C C

当我们从**S**的第一个规则开始，一切都进展顺利并且生成了终结符**x**。但是当我们从**S**的第二条规则（规则2）开始时，我们就被卡住了，而当我们从规则3开始时，我们就发现进入了一个无限循环中，生成越来越多的**C**。规则2、3和5永远都不会产生一个成功的生成过程：他们是无意义的规则，并且也无法在不影响语言生成的情况下从语法中移除掉。

无用的规则并不是根本性的问题：它们不会妨碍正常的生成过程。尽管如此，它们依旧是语法中的枯木，而总有人会想移除它们。并且，当他们出现在由程序员指定的语法中时，它们可能会指向某些错误，那就会想要检测它们并给出警告或错误信息。

上述语法的问题很容易理解，但可以表明，大多数情况下是很难判定在0型或1型语法中的一条规则是无用的：不可能有一种算法能在所有情况下都正确判断。然而，对上下文无关语法来说，这个问题就是相当容易解决的了。

在上下文无关语法中的规则可能是无用的，因为三个原因：它们可能包含未定义的非终结符，从起始符号开始可能无法到达它们，以及它们可能无法生存任何东西。接下来我们会详细讨论这些缺陷；2.9.5节给出了一个算法可以帮助语法拜托这些缺陷。

2.9.1 未定义的非终结符

一些规则的右侧可能包含一个非终结符，其没有对应的生成规则。这样的规则永远也不会出现问题，并且可以被从语法中移除。如果我们这样做，我们可能就移除了另一个非终结符的最后一个定义，而这个非终结符会反过来又变成未定义的，等等。

我们会进一步看到（例如4.1.3节），认可未定义非终结符偶尔也是有意义的。在它们右侧的解释它们的规则，又是可以删除的。

2.9.2 不可到达的非终结符

如果一个非终结符，无法从起始符号到达它，它的定义规则将永远不会被用到，那它也不能为任何句子的生成过程做出贡献。不可到达的非终结符有时也被称为“不被使用的非终结符”。但这个说法有点误导，因为一个不可到达的非终结符*A*也可能出现在右侧*B-->...A...*，使它看起来是有用的而让*B*成为不可到达的；这同样也适用于*B*，等等。

2.9.3 非生成性规则和非终结符

假设X有其唯一规则$X \rightarrow aX$并且假设从起始符号可以到达X。那现在X依然不会对其语言语法中的句子做出贡献，因为一旦引入的X就没有办法摆脱它了：X是一个非生成性的非终结符。此外，任何规则其右侧含有X的都是非生成性的。简单说，任何本身不产生非空子语言的规则是非生成性的。如果一个非终结符的所有规则都是非生成性的，那这个非终结符就是非生成性的。

一个极端的例子就是一个语法中的所有非终结符都是非生成性的。这种情况发生在一个语法的所有右侧都包含至少一个非终结符。然后就没有办法能摆脱非终结符，那这个语法就是非生成性的。

这三种情况合起来就被成为无用的非终结符。

2.9.4 循环

上述定义使所有可以包含在一个句子生成过程中的规则是“非无用”的，但仍有一类规则并不是真的有用：类似$A \rightarrow A$ 形式的规则。这类规则被称为循环。循环也可以是间接的：$A \rightarrow B, B \rightarrow C, C \rightarrow A$；而且它们可以被隐藏：$A \rightarrow PAQ$，$P$□ ε, Q□ ε，所以生成过程$A \rightarrow PAQ \rightarrow ...A... \rightarrow A$也是可能的。

一个循环可以合理的而出现在句子的生成过程中，并且如果它确实出现了，依旧会有这个句子的另一个不包含循环的生成过程。循环不对语言最贡献，任何句子的生成过程包含一个循环的无限模糊的，意味着对它来说有无限多的生成树。4.1.2 节中给出了循环检测算法。

不同解析器对带有循环的语法的反应不一样。有些（大部分的普通解析器）诚实的试图去构建无限数量的推导树，另一些（例如，CYK解析器）如上文所述一样将循环折叠起来，还有一些（最具决定性的解析器）拒绝了这样的语法。而ε规则可以隐藏循环加剧了这个问题：一个循环只有当某些非终结符生成了ε时，才是可见的。

一个不含有无用非终结符和循环的语法，才被称为一个正确的语法。

2.9.5 清理上下文无关语法

通常情况下，人们提供的语法不会包含未定义，不可到达或非生成性的非终结符。如果出现了，那几乎可以肯定是一个失误（或者是用来测试的！），然后我们就要检测和报告出来。然而，这种异常情况很容易出现在生成的语法或由其他语法转换所引入的语法中，这种情况下我们就希望能检测到然后“清理”一下语法。清理语法也是十分重要的，当我们获取解析森林语法的解析结果时（3.7.4节，13章以及其他很多地方）。

从一个上下文无关语法中检测和删除无用非终结符和规则的算法包含两个步骤：移除非生成性规则以及移除不可到达的非终结符。令人惊叹的由于未定义的非终结符，似的移除无用的规则并不是必须的：第一步为我们自动完成了这个过程。

```
Ss  -->  A B | D E
 A  -->  a
 B  -->  b C
 C  -->  c
 D  -->  d F
 E  -->  e
 F  -->  f D
```

Fig. 2.27. A demo grammar for grammar cleaning

我们将使用图Fig 2.27中的语法来进行演示。它看起来相当单纯：它的所有非终结符都是定义了的，而且它也没有表现出任何可疑的结构。

2.9.5.1 移除非生成性规则

我们通过找到一个生成性规则来寻找非生成性的规则。我们的算法取决于观测，如果一个规则的右侧包含的符号都是生成性的则这个规则就是生成性的。终结符是生成性的因为它能生成终结符，空也是生成性的因为它能

生成空字符串。如果一个非终结符有一个生成性规则对应于它那它也是生成性的，但问题是起初我们并不知道哪条规则是生成性的，因为这本身就是我们在努力寻找的。

我们解决这个问题，首先通过使所有规则和非终结符都是“不知道”的。现在我们来看图Fig 2.27的语法，对于每一条我们不知道的规则，其右侧的成员都是生成性的，那我们就标记这条规则和它定义的非终结符为“生成性”的。这将为规则**A--->a, C--->c,** 以及**E--->e**产生标记，还有非终结符**A, C**和**E**。

现在我们知道的更多了，并且可以将这些用于对语法的第二轮扫描了。这使我们能标记规则**B--->bC**以及非终结符**B**，因为现在**C**已知是生成性的了。第三轮确定了**S--->AB**和**S**。第四轮没有产生新的东西，所以也就没有进行第五轮的必要了。

现在我们知道**S, A, B, C**和**E**是生成性的，但是**D**、**F**以及规则**S--->DE**还是标记“不知道”的。然而我们知道了更多的：知道我们尝试了生成性的所有可能路径，并且没有为**D**、**F**以及**S**的第二条规则找到任何可能的路径。这意味着我们现在可以更新一下对于“非生成性”的“不确定性”的信息了。**D**、**F**的规则以及**S**的第二条规则可以从语法中移除了；结果如果Fig 2.28所示。这就使得**D**、**F**成为了未定义的，而**S**仍然留在语法中因为它是生成性的，虽然有一个非生成性规则。

$S_s \rightarrow A\ B$
$A \rightarrow a$
$B \rightarrow b\ C$
$C \rightarrow c$
$E \rightarrow e$

Fig. 2.28. The demo grammar after removing non-productive rules

看看当语法中包含一个未定义的非终结符，例如*U*，将会发生什么，是一件有趣的事情。首先*U*将被预定义为“不知道”的，而因为没有规则定义它，它将一直保持“不知道”状态。因此，任何右侧有着*U*的规则*R*都将会是“不知道”的。最终两者都会被定义为“非生成性”的，然后所有的*R*规则都会被移除。可以看到“未定义的非终结符”只是“非生成性”的非终结符的一种特殊情况：因为它没有规则，所以他是非生成性的。

上述知识改进的算法使我们关于闭包算法的第一个例子。闭包算法有两个主要特点：初始化，是对最初了解的一个评估，部分源于其现状和“不知道的”部分；推导规则，介绍从几个地方获取的信息是如何结合的。对于我们的问题的推导规则是：

对于每一个我们知道其右侧的所有成员都是生成性的规则，标记它定义的规则和非终结符为“生成性”的。

推导规则一直重复直到没有不在有任何变化，这一点在闭包算法中是隐式的。然后初步的“不知道”类型就会变成一个更明确的“非X”，“X”是算法被设定来检测的属性。

因为预先知道所有依旧是“不知道”状态的最终都将会变成“非X”，所以许多闭包算法的描述和实现直接跳过整个“不知道”步骤，而直接初始化所有的为“非X”。在执行中，这不会有太大差别，因为计算机内存中位的意义不是在计算机中而是在程序员脑中，但是在打印书本描述中这种做法就是不优雅的也是让人疑惑的，因为初始化语法中的所有非终结符为“非生成性”是不正确的。

本书中我们会看到很多闭包算法的例子；在3.9节中会有详细的讨论。

2.9.5.2 移除不可到达的非终结符

一个非终结符存在至少一个句子形式就可以称为是可到达的或可访问的，从开始出现的起始符号开始。所以如果对一些α和β存在S□$\alpha A\beta$那么非终结符A就是可到达的。

我们通过找到“生成性”的规则和非终结符来找到非生成性的那些。同样的，我们通过找到可到达的非终结符来找到不可到达的那些。为此，我们可以使用以下的闭包算法。首先，起始符号被标记为“可到达的”；这就是初始化。然后，语法中每一个标记了A的$A \to \alpha$形式的规则，α中所有的非终结符都会被标记；这就是推导规则。我们持续应用推导规则直到不再有变化产生。现在剩下的未标记的非终结符就是不可到达的，而他们对应的规则可以被删除了。

第一轮标记**A**和**B**；第二轮标记**C**，第三轮没有任何不变化。结果就是--一个干净的语法--见图Fig 2.29。如图Fig 2.27中可到达和生成性的规则**E--->e**，子啊移除非生成性规则后变成了孤立的，然后被第二步的清理算法给删除掉了。

```
Ss  →  A B
A   →  a
B   →  b C
C   →  c
```

Fig. 2.29. The demo grammar after removing all useless rules and non-terminals

移除不可到达的规则不会导致在一个可到达的规则中使用的非终结符**N**变成未定义的，因为只有当移除了**N**的所有定义的规则才能变成未定义的，但是又因为**N**已经是可到达的，所以上诉过程将不会移除它的任何一条规则。对相同参数的稍微修改可以看出移除不可到达的规则不能导致一个在可到达规则中使用的非终结符**N**变成非生成性的：生成性的**N**，否则无法在前面的清理步骤中留存下来，只能通过移除它的定义规则来使之成为非生成性的，但是由于**N**是可到达的，上面的过程将不会移除它的任何规则。这最后表明在移除了非生成性的非终结符以及移除了不可到达的非终结符之后，我们没有必要再做一遍移除非生成性的非终结符。

然而有趣的是请注意，如果先移除不可到达的非终结符然后在移除非生成性的规则将有可能导致语法再次含有不可到达的非终结符。图Fig 2.27的语法是一个例子。

此外需要注意的是，清理一个语法可能回移除所有的规则，包括其实符号的规则，描述空语言的语法就是例子；见2.6节。

移除非生成性规则是一个自底向上的过程：只有底层，终结符所处的位置，才能知道哪些是生成性的。移除不可到达的非终结符是一个自顶向下的过程：只有顶层，起始符号所处的位置，才知道哪些是可到达的。

2.10 设定上下文无关和正则语言的属性

由于语言是集合，所以很自然会问到集合的标准操作——并集、交集和补集（补充）——是否能用在语言上，如果可以，要怎么做。

S^1和S^2的并集包含两两个集合中的全部元素；写作$S^1 \cup S^2$。交集包含了两个集合中的共同元素；写作$S^1 \cap S^2$。S的补集包含了属于Σ^*但不属于S的元素；写作 S。在正式语言的上下文中，这些集合是通过语法定义的，所以实际上我们是想要对语法进行操作，而不是语言。

为两种语言的并集构建语法，对上下文无关和正则语言来说是繁琐的（实际上对所有Chomsky类型都是）：仅仅构建一个新的起始符号$S' \rightarrow S^1 | S^2$，S^1和S^2是描述两种语言的语法的起始符号。（当然，如果我们想结合两种语言成为一种新的，那我们必须确保他们之中的名字是不同的，不过这是很容易的事情。）

然而交集是另一回事，因为两个上下文无关语言的交集并不一定是上下文无关的，如以下示例所示。有两个语言$L1 = a_n b_n c_m$和$L2 = a_m b_n c_n$，他们由CF语言描述

$$
\begin{array}{lll}
L_{1s} & \rightarrow & \mathtt{A\ P} \\
\mathtt{A} & \rightarrow & \mathtt{a\ A\ b} \mid \varepsilon \\
\mathtt{P} & \rightarrow & \mathtt{c\ P} \mid \varepsilon
\end{array}
\quad \text{and} \quad
\begin{array}{lll}
L_{2s} & \rightarrow & \mathtt{Q\ C} \\
\mathtt{Q} & \rightarrow & \mathtt{a\ Q} \mid \varepsilon \\
\mathtt{C} & \rightarrow & \mathtt{b\ C\ c} \mid \varepsilon
\end{array}
$$

当我们拿到一个同时属于两个语言的字符串放入交集时，就有了$a_p b_q c_r$这种形式，而由于L^1和L^2其中$p = q$以及$q = r$。所以交集中包含$a_n b_n c_n$这样形式的字符串，而我们知道这样的语言不是上下文无关的（见2.7.1节）。

CF语言的交集有一些奇特的属性。第一，两个CF语言的交集总是有一个1型语法——但这个语法却不容易构建。更值得注意的是，三个CF语言的交集比两个的交集要强大的多：Liu 和 Weiner[390]表明，可以获得k个CF语

言的交集，而不是*k-1*个。除此之外，任何1型语言，甚至任何0型语言，可以通过两个CF语言的交集来构建，我们就能擦除结果字符串中的所有属于可擦除符号集中的符号。

我们将用来演示这个惊人现象的CS语言，是由两个相同部分的句子组成的集合：*ww*，*w*是给定字符集中的任何字符串；例如**aa**和**abbababbab**。用来相交的两个语言由以下定义：

```
L3s  →  A P                                   L4s  →  Q C
 A   →  a A x | b A y | ε        and           Q   →  a Q | b Q | ε
 P   →  a P | b P | ε                          C   →  x C a | y C b | ε
```

其中**x**和**y**是可擦出符号。第一个语法生成的字符串由三个部分组成，**a**和**b**的序列A^1，其次是其“黑暗镜像”M^1，**a**对应**x**，**b**对应**y**，再接着是**a**和**b**的一个任意序列G^1。第二个语法生成的字符串包含，**a**和**b**的一个任意序列G^2，一个“黑暗”序列M^2以及它的镜像A^2，其中再次**a**对应**x**，**b**对应**y**。交集强制*A1 =G2, M1 =M2,and G1 =A2*。这使得A^2成为了A^1镜像的镜像，同理A^1也是这样。交集中的一个句子示例如**abbabyxyyxabbab**，我们可以看到其镜像**abbab**和**yxyyx**。现在我们擦除可擦除符号**x**和**y**，就得到了最后的结果**abbababbab**。

通过使用应用上述镜像，就能够很简单的证明任何0型语言能够通过两个CF语言的交集，加上一组可擦除符号来构建了。详细介绍，见Révész [394]。

注意，一个上下文无关和一个正则语言的交集，一般都是一个上下文无关语言，并且，有一个相关的简单算法来为这个交集语言构建一个语法。这让非凡的解析算法得到了增长，浙江在13章进行讨论。

如果我们不能得到两个CF语言的交集，并且仍处于CF语言中，那我们肯定不能得到一个CF语言的补集并仍在CF语言中。如果我们能得到，我们就能得到两个语言的补集，让后取其交集然后在取其补集，最后就得到它们的补集。公式：$L^1 \cap L^2 = \square((\square L^1) \cup (\square L^2))$；这个公式就是大家熟知的De Morgan定律（De Morgan’s Law）。

在5.4节我们将会看到正则（3型）语言和正则语言的并集，交集和补集。

有趣的是推测一下将会有什么发送，如果正式语言是基于集合理论，一开始就使用所有的集合操作，而不是Chomsky层次理论。那么上下文无关语言还会被发明么？

2.11 语义连接

有时解析只服务于检测一个字符串的正确性；字符串符合给定的语法就是我们想要知道的，例如因为它证实了我们某些观察模式的猜想，那确定就是被我们特意为之设计的语法正确描述的。然而，一般情况下，我们想要做的更多：我们知道字符串都传达一种含义，一种语义，而且这种语义直接与这个字符串的生成树的结构相关。（如果不是，那我们就弄错了语法！）

语法附加语义是通过一个非常简单但有效的方式完成的：语法中的每一个规则，其规则右侧成员语义相关的一个语义子句被附加在其左侧的语义上，这种情况下语义从生成树的节点直接流向起始符号；或者反过来说，这种情况下，语义反向从起始符号流向节点；又或者同时的，在这种情况下，语义信息一会向上一会向下流动，直到达到某个稳定状态。语义信息向下的流动称为继承：生成树上的每个规则从其父节点继承语义。语义信息向上流动称为派生：每一个规则由其子节点派生。

有许多方式来表达语义子句。因为我们的主题是解析和语法而不是语义，我们将只会简要描述两个经常用到并得到充分研究的技术：属性语法和转导语法。我们会使用同一个简单例子来解释，一位数数字的总和的语言；这个语言的句子的语义是总和的值。这个语言由图Fig 2.30的语法生成。例如他的一个句子是3+5+1；其语义就是9。

```
1.  Sum_S  ->  Digit
2.   Sum   ->  Sum + Digit
3. Digit   ->  0 | 1 | ··· | 9
```

Fig. 2.30. A grammar for sums of one-digit numbers

2.11.1 属性语法

属性语法中的语义子句假设生成树的每一个节点都有空间给一个或多个属性，这些属性就是放在生成树节点上的值（数字，字符串或其他东西）。为简单起见，我们限定这个属性语法每个节点只有一个属性。这种语法中一个规则的语义子句包含一个公式，用以计算一个规则(由生成树的节点展示)中的一些非终结符着这个规则的其他终结符。这些语义动作之和规则的本地属性关联：整体的语义由所有本地计算的结果构成。

如果规则*R*的语义子句计算*R*左侧的属性，这个属性会派生。如果它计算*R*的右侧非终结符的属性，例如*A*，那这个属性就被*A*继承。派生属性也被称为“合成属性”。我们例子的属性语法是：

1.	$\mathtt{Sum}_S$	→	Digit	$\{A_0 := A_1\}$
2.	Sum	→	Sum + Digit	$\{A_0 := A_1 + A_3\}$
3a.	Digit	→	0	$\{A_0 := 0\}$
	···	···		
3j.	Digit	→	9	$\{A_0 := 9\}$

花括号中给出了语义子句。A^0是左侧的（派生）属性；A^1，...，A^n是右侧成员的属性。通常来说，右侧的终结符也计入*A*的索引，虽然它们（通常）不携带属性；规则2的**Digit**在位置3并且其属性值为A^3。用于处理属性语法的大多数系统都少有重复的方式来通过3j表达3a。

3+5+1 的初始生成树在图Fig 2.31中给出。首先只有节点的属性的明确的，但是到生成树的一个右侧的所有属性都明确后，我们就能使用它的语义子句来计算它的左侧了。这种方式下，属性值（语义）渗透到树上，最后到达起始符号，并提供给我们全句的含义，如图Fig 2.32所示。属性语法是非常强大的操控语言语义的方式。这些将会在15.3.1节中详细讨论。

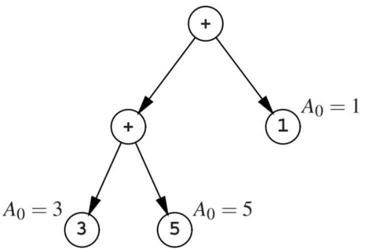


Fig. 2.31. Initial stage of the attributed production tree for **`3+5+1`**

2.11.2 转导语法

转导语法将一个字符串（“输入字符串”）的语义定义为另一个字符串，“输出字符串”，而不是起始符号的最终属性：

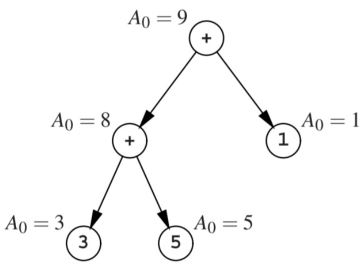


Fig. 2.32. Fully attributed production tree for `3+5+1`

这种方法没那么强大，但比起属性语法却更简单并且也够用了。生成规则中的语义子句只是应该在对应节点输出的字符串。我们假设一个节点的字符串紧接着它所有子节点的字符串后输出。其他的变种是可能的并且很正常。我们现在可以编写一个转导语法，其将数字之和转换为计算机指令之和。

```
1.    Sum_S  ->  Digit          "make it the result"
2.    Sum    ->  Sum + Digit    "add it to the previous result"
3a.   Digit  ->  0              "take a 0"
       ···   ···
3j.   Digit  ->  9              "take a 9"
```

这个转导语法将**3+5+1**转换为：

```
take a 3
make it the result
take a 5
add it to the previous result
take a 1
add it to the previous result
```

这就是**3+5+1**真实的“意义”。

2.11.3 增广转换网络

语义可以被引入一个递归过渡网络（2.8节），通过附加动作到图形的转换中。这些动作可以设置变量，构造数据结构，等。因此增强的递归转换网络被称为增广过渡网络（或*ATN*）（Woods [378]）。

2.12 语法类型的隐喻比较

教科书声称“n型语法比n+1型语法更强大，n = 0，1，2”，并且经常可以读到这样的语句“一个正则（3型）语法不够强大以匹配括号内的”。有趣的是，看看到底是什么样的力量。天真，一个人可能以为它的力量是生成越来越大的集合，但这明显是不正确的：最大的可能的集合Σ^*，可以很容易由3型语法生成：

S^s --->[Σ]S| ε

[Σ]是语言中符号的缩写。只是当我们想要限制这个集合时，我们就需要更强大的语法。更强大的语法可以在正确和不正确的句子间定义更复杂的边界。有的边界定义的太好导致没有任何语法能描述它（也就是，通过任何生成过程）。

这个想法在图Fig 2.33中进行了比喻描述，图中一支玫瑰由越来越细的轮廓接近。在这个比喻中，玫瑰代表语言（想象语言中的句子就是玫瑰的分子）；语法就是为了描绘它的轮廓。一个正则语法只允许我们用水平直线和垂直线段来描绘花朵；直尺和T型尺就够了，但结果只是一个粗陋和机械的图片。CF语法能通过各个角度的直线和圆弧近似描绘；图画还是可以使用传统的圆规和直尺就够了。最终结果很生硬但好歹能辨认了。CS语法可以让我们用平滑的曲线包围花朵了，但是曲线太平滑了：它无法表现尖角，而且偏离了复杂的交点；不过，依旧有了非常逼真的效果图。不受限制的短语结构语法可以完美展现大概轮廓。一朵玫瑰不可能被限定在一种限定的描述中；其本质永远都是我们嗦无法企及的。

一个更简单更有效的例子，可以在一个能通过多种语法类型生成的Java[1]程序的继承集合中找到。

- 所有词法正确的Java程序可以通过一个正则语法生成。一个Java程序词法是正确的，如果字符串中没有换行符，评论在文件结尾时被终止，且所有数值常量都是正确形式，等等。

- 所有语法正确的Java程序都可以通过一个上下文无关语法生成。这些程序在理论上都符合（CF）语法。
- 所有语义上去的Java程序都可以通过一个CS语法生成。这些都是通过一个Java编译器没有抛出错误信息的程序。
- 在有限时间内运行给定输入会终止的所有Java程序可以通过一个无限制的短语结构语法生成。然而，这种语法会非常复杂，因为它会包含Java库的进程和Java运行时间系统的详细描述。
- 解决给定问题（例如，下棋）的所有Java程序不能通过一个语法（尽管集合的描述时有限的）生成。

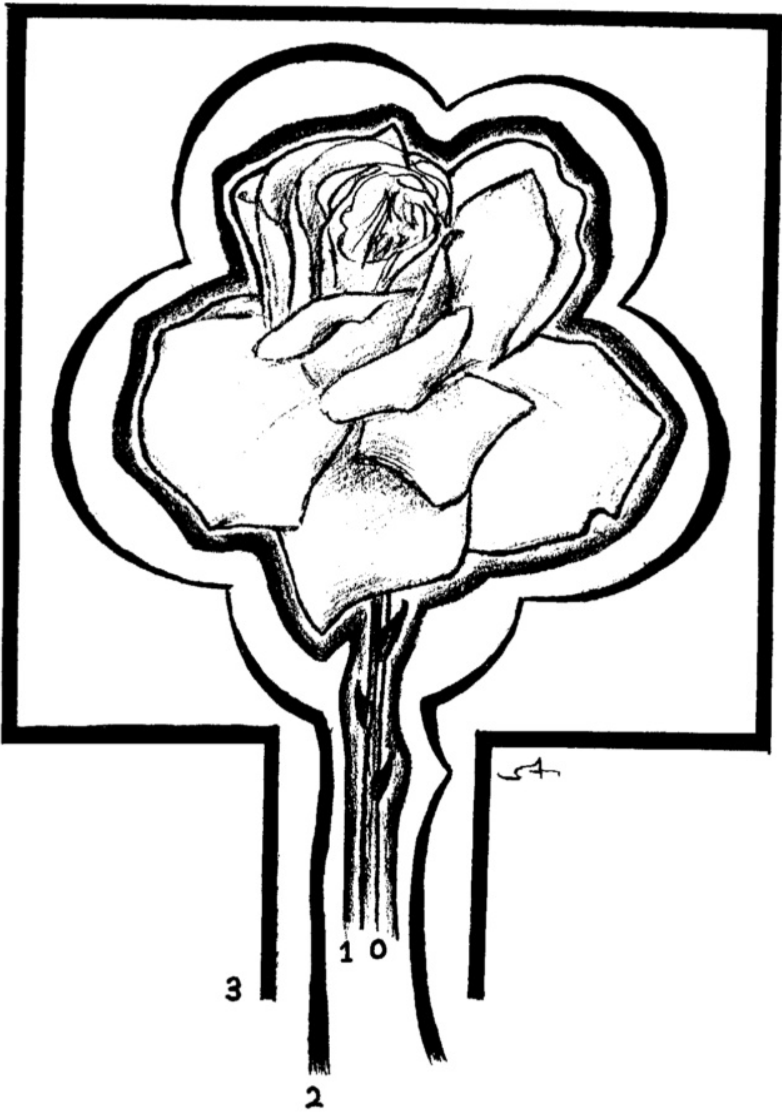


Fig. 2.33. The silhouette of a rose, approximated by Type 3 to Type 0 grammars

> 1. 我们在这里使用编程语言Java，因为我们希望读者或多或少能熟悉它。对于手册所给一个CF语法的，任何一种编程语言都可以做到。↩

2.13 总结

Chomsky语法是一种有限的机制，产生通常是无限集合的字符串，一门“语言”。不同于其他许多集合生成机制，这种生成过程指派一个结构给生成的字符串，可以用来向其附加语义。对于上下文无关语法（2型），这种结构是一棵树，允许语义由分支的语义组成。这是上下文无关语法的重要性的基础。

问题

问题**2.1**：2.1.3.4节的对角化似乎是一个不在列表上语言的有限描述。为什么这个描述不在列表上，列表可是包含所有有限描述的？

问题**2.2**：在2.1.3.4节中，我们考虑函数*n,n+10*和*2n*，来找到应该有别于行*n*的位的位置。这些函数的一般形式是什么，既，什么样的函数集合可以生成不具有有限描述的语言？

问题**2.3**：位Manhattan龟路径设计一个语法，使其从其起点开始不允许向西爬行。

问题**2.4**：图Fig 2.7的单调1型语法生成所有$a^n b^n c^n$形式的字符串，*n*≥*1*。为什么*n=0*排除在外？

问题**2.5**：设计一个1型语法，可以生成包含两个相同部分的所有字符串的语言：*ww*，*w*是给定字母表（见2.10）的任何字符串。

问题**2.6**：在2.4.1节，我们有句子生成机制，将新创建的句子形式添加到队列结尾，并声称这实现了广度优先生成。当我们将它放在队列开头时，机制采用深度优先生成。说明这是不起作用的。

问题**2.7**：2.4.1节的最后一段中说到“在增加（实际上是‘不减少’）的长度”。解释为什么说是“不减少”就足以表明。

问题**2.8**：将一个没有递归的语法生成的有限语言的字符串数量与该语法的结构关联。

问题**2.9**：查阅2.6节。在你的计算环境中找到更多的例子，零作为一个数字得到二等对待。

问题**2.10**：在你最喜欢的解析器生成器系统中，为语言{ε}写一个解析器。同样也为语言{}写一个解析器。

问题**2.11**：使用*uvw*理论（2.7.2节），说明对于语言$a^i b^i$没有可用的3型语法。

问题**2.12**：在2.9节我们说到，无用的规则可以被从语法中移除掉而不影响语言的生成。这似乎是表明“其移除不影响语言”就是我们所希望的，而不是仅仅是没有无用的。注释。

问题**2.13**：根据2.2.2节，写一个Chomsky生成过程，作为一个闭包算法。

3 解析简介

根据语法来分析字符串意味着重建生成树，以展示给定字符串是如何通过给定语法生成的。在这方面它有重要意义，第一个解析方面的出版物（Greibach的1963年博士论文[6]）,命名为“逆向短语结构生成器”（Inverses of Phrase Structure Generators），而短语结构生成器被理解为一个从短语结构（事实上是上下文无关）语法生成短语的系统 。

虽然基于0型或1型语法生成句子产生的是生成图而不是生成树，并因此解析得到了解析图，我们应该使用2型，上下文无关语法，以专注于解析以及最后的解析树。偶尔我们会触及基于0型或1型语法的解析，例如3.2节，只是为了展示它是一个有意义的概念。

3.1 解析树

关于重建解析树有两个重要的问题：我们为什么需要它；我们如何实现它。

恢复生成树的需求是不自然的。毕竟，语法是一组字符串的凝聚态描述，既语言，并且也许我们的输入字符串可能属于或者不属于这个语言；没有涉及内部结构或生成路劲。如果我们坚持这个正式观点，我们可以问的唯一有意义的问题就是，一个给定的字符串是否可以被一个语法识别；任何关于如何做的问题都是无意义的标志，甚至只能是一种好奇心的表示。然而在实践中，附加了语义的语法：特定的语义附加到特定的规则，而且为了确定一个字符串的语义，我们需要找出在参与了其生成过程的规则以及是如何参与的。总之，识别是不够的，我们还需要恢复生成树以得到句法方式的全部优势。

被恢复的生成树称为解析树。在0型和1型语法中，将语义附加到特定规则几乎是不可能的事实说明了它们在解析中的微不足道，相比于2型和3型语法。

如何重建生成树是本书余下部分的主题。

3.1.1 解析树的大小

一个有*n*标记的字符串的解析树由属于终结符的*n*个节点，在加上大量属于非终结符的节点组成。令人惊讶的是，不能有多于$C_G n$的属于非终结符的节点，在*n*标记节点的一个解析树中，C_G是由语法决定的一个常量，证明语法没有循环。这意味着，任何解析树的大小都是线性的，根据输入的长度。

表明确实需要用一系列步骤来完成。我们首先证明对语法来说所有右侧都是长度2；这就导致了二叉树，树上每个节点要么有两个子节点要么就是叶子（没有子节点的节点）。二叉树有着显著的特点，所有的给定树叶数量的二叉树都有同样数量的节点，而不在乎它们的形状。下面我们看右侧长度>2的语法，接着有单元规则的语法，最后是可为空规则的语法。

正如我们说的，长度为*n*的输入字符串包含*n*标记的节点。当解析树尚不存在时，这些节点是没有父节点的叶子。现在我们来构建一棵二叉树来给每个叶子一个父节点，父节点都标记了来自语法的非终结符。第一个父节点P_1我们添加的，减少了2个没有父节点的叶子，但是现在P_1自身成为了没有父节点的节点；所以我们现在有$n+1$个节点，其中$n-2+1 = n-1$个节点没有父节点了。同样的事情发生在添加的第二个父节点P_2上，不论P_1是否是其子节点；所以现在我们有$n+2$个节点，其中$n-2$个没有父节点。*j*步之后，我们就有了$n+j$个节点，其中$n-j$个没有父节点，并且在$n-1$步之后，我们就有$2n-1$个节点，其中1个没有父节点。那没有父节点的1个节点就是根节点，然后解析树就完成了。所以我们看到，当所有右侧长度是2时，对于输入长度是*n*的解析树，包含$2n-1$个系欸但，其线性在*n*。

如果有的右侧的长度>2，那可能只需要更少的父节点来构造这颗树。所以整棵树的大小可能会小于$2n-1$，并且肯定会小于$2n$。

如果语法包含单元规则 - $A \rightarrow B$形式的规则 - 那么添加父节点会减少无父节点数这点就不对了：当一个无父节点的节点*B*通过规则$A \rightarrow B$获得了一个父节点*A*，它就不再是无父节点的节点了，但是*A*又成为了无父节点的节点，并且更糟的是，节点数却增加了一个。并且可能有必要重复这个过程，就

是说使用规则$Z \to A$，等等。但是最终端元规则的链必定会走到尽头，比如P（所以我们有了$P \to Q \cdots Z \to A \to B$），或者语法中就有一个环。这意味着$P$获得了一个有多个子节点的父节点，然后无父节点的节点数减少了（或者P就是根节点）。所以单元规则能做的最糟糕的事情就是“延长”每一个节点通过构造因子C_u，因此解析树的大小会小于$2C_u n$。

如果语法包含形式$A \to \varepsilon$这样的规则，只有数量有限的ε能在输入的相邻标记中被识别，或者语法中就又有一个环了。所以可为空规则能做的最糟糕的事情就是“延长”输入通过构造一个因子C_n，在两个标志之间被识别的ε的最大数目，还有解析树的大小都小于$2C_n C_u n$，其线性大小是n。

另一方面，如果语法允许包含循环，以上两个过程就将会在解析树中引入无限延伸的节点，那树的大小就是任何大小了。

3.1.2 各种模糊性

一个语法产生的句子可以很容易拥有多于一个的生成树，既，很容易有多于一种的方式来生成一个句子。从正式角度来看，这不是大问题（一个集合不会计算它包含了一个元素多少次），但是只要我们对语义感兴趣，那这差别就很重要了。不足为奇的是，拥有多个生成树的的句子被称为模糊性，但是我们必须立即区分本质的模糊性和貌似的模糊性。差异来源于我们对生成树本身并不感兴趣的事实，而更感兴趣于它们所描述的语义。一个歧义句是貌似模糊性的，如果它的所有的生成树都描述同样的语义；如果部分语义不不同，其模糊性就是本质的。“模糊”的概念也可以用来定于语法：语法是本质模糊的如果它可以生成一个本质模糊的句子，是貌似模糊的如果它能生成一个貌似模糊的句子（但不是一个本质上模糊的），以及是非模糊性的如果都不能生成的话。一个语法的可能模糊性测试，间9.14节。

图Fig 3.1给出了一个简单的模糊性语法。注意规则2不同于图Fig 2.30中的。

1.	$\texttt{Sum}_s$	→	`Digit`	$\{ A_0 := A_1 \}$
2.	`Sum`	→	`Sum + Sum`	$\{ A_0 := A_1 + A_3 \}$
3a.	`Digit`	→	`0`	$\{ A_0 := 0 \}$
	···			
3j.	`Digit`	→	`9`	$\{ A_0 := 9 \}$

Fig. 3.1. A simple ambiguous grammar

现在**3+5+1**有了两个生成树（图Fig 3.2），但是两者的语义是一样的：9。模糊性是不确定的。然而如果我们把**+**变成**-**，那其模糊性就是本质的了，如图Fig 3.3。图Fig 2.30的非模糊性语法，当**+**变成**-**后，依旧是非模糊性，并且保持正确的语义。

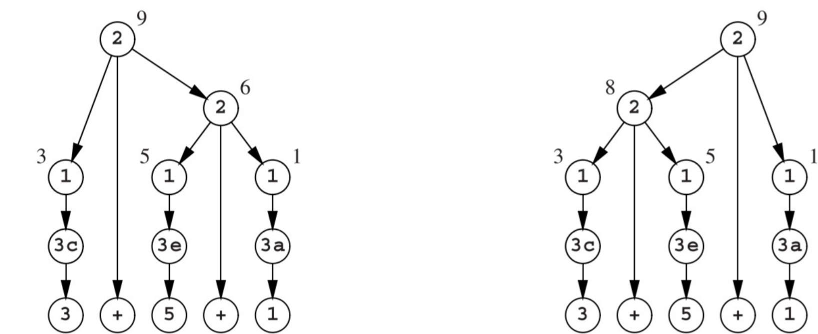


Fig. 3.2. Spurious ambiguity: no change in semantics

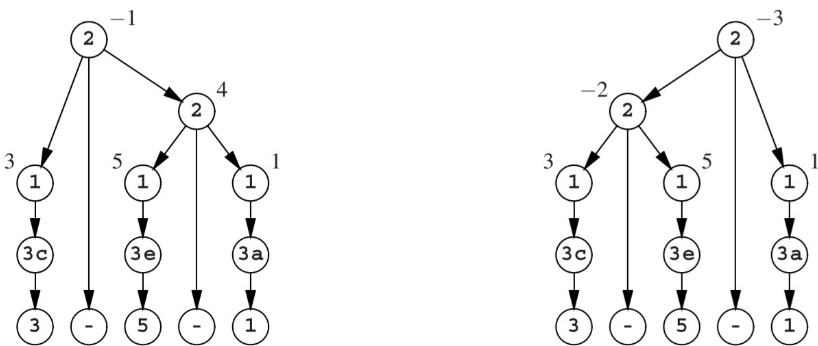


Fig. 3.3. Essential ambiguity: the semantics differ

奇怪的是，语言也可以是模糊性的：（上下文无关）语言，其语法没有非模糊性的。这样的语言是固有模糊性的。语言$L = a^m b^n c^n \cup a^p b^p c^q$就是一个例子。*L*中的句子要么由一些**a**后接着一组嵌套的**b**和**c**的序列组成，要么由一些**c**后接着一组嵌套的**a**和**b**的序列组成。例句：**abcc**，**aabbc**，还有**aabbcc**；**abbc**是一个非句子例子。*L*由图Fig 3.4的语法生成。

S_s → AB | DC
A → a | aA
B → bc | bBc
D → ab | aDb
C → c | cC

Fig. 3.4. Grammar for an inherently ambiguous language

直观的说，为什么L是模糊性的是相当清晰的：语法的任何可以生成$a^m b^n c^n$的部分都无法避免生成$a^n b^n c^n$，并且任何可以生成$a^p b^p c^q$的部分都无法避免生成$a^p b^p c^p$。所以无论我们做什么，与a，b，c数字相等的形式将总是会被生成两次。但真实的证明确实无法做到，这不是本书范围之内的东西了。

3.1.3 解析树的线性化

通常构建一棵实际的解析树是不方便也是不必要的：相反一个解析器可以生成一列规则编号，这意味这其线性化了解析树。有三种主要方法来线性化一棵解析树，前缀、后缀和中缀。前缀表示法中，每一个节点都被列出通过列出其编号然后接着是子节点列表的前缀，由左到右的顺序；这给了我们最左侧的推导（图Fig 3.2的右边的树）：

最左侧:2 2 1 3c 1 3e 1 3a

如果一棵解析树是根据这个方法构建的，那它是在前序中构建。在前缀表示中，每一个节点被列出通过列举全部前缀表示按从左到右的顺序，后面接着节点自身规则的编号；这给了我们最右侧的推导（同一棵树的）：

最右侧: 3c 1 3e 1 2 3a 1 2

这在后序中构建解析树。中缀表示法中，每个节点都被列举，通过首先给一个列举在括号中第一个n子节点的中缀，然后接着是节点的规则编号，然后接着是一个列举在括号中其余子节点的中缀；n可以自由选择，甚至规则之间可以不同，但是$n = 1$ 是正常的。中缀在派生中是不常见的，但偶尔是有用的。$n = 1$的情况被称为左角推导；在例子中我们得到：

左角: (((3c)1) 2 ((3e)1)) 2 ((3a)1)

中缀表示法需要括号来使我们从中重建生成树。最左和最右推导不用括号也能完成，使我们准备好语法来找到每个节点的子节点数。

很容易区分一个推导是最左还是最右的：一个最左推导开始于起始符号的一个规则，而一个最右推导开始于只生成终结符的一个规则。（如果两个条件同时成立，那只有一个规则可以，可以同时是最左和最右推导的。）

几个不同推导的存在不应该与模糊性混为一谈。不同的推导只是同一个生成树的符号变形。对于其不同没有不同的语义可以附加。

3.2 解析一个句子的两种方法

一个句子和其起源的语法之间的基本联系就是解析树，解析树描述了语法是如何生成一个句子的。为了这种连接的重建，我们需要一种解析技术。当我们查阅解析技术方面的文档时，我们似乎找到了几十个，不过却只有两个是解析方面的；其余的都是技术细节和点缀。

第一种方法尝试去模仿初始生成过程，通过重新推导从起始符号推导出句子。这种方式被称为自顶向下，因为解析树是从上向下重构建的。[1]

第二种方式试图回滚生成过程并减少回到起始符号的判定。自然的这种方式就被称为自底向上。

> [1]. 在计算机中，树的生长过程是从根向下生长的；这与物理学中有一个负电荷的点子相类似。↩

3.2.1 自顶向下解析

假设我们有图Fig 2.7中语言$\mathbf{a^n b^n c^n}$的单调语法，这里重新提一下：

$$
\begin{aligned}
S_S &\rightarrow aSQ \\
S &\rightarrow abc \\
bQc &\rightarrow bbcc \\
cQ &\rightarrow Qc
\end{aligned}
$$

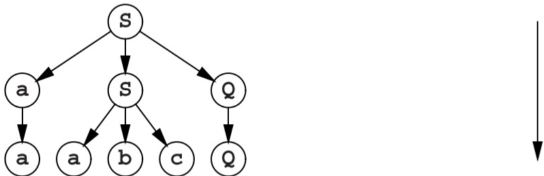
并且假设（输入的）句子是**aabbcc**。首先我们尝试一些自顶向下解析方式。我们知道生成树必须从起始符号开始：



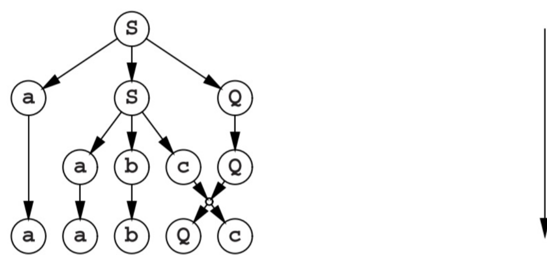
那现在第二步是什么？对于**S**我们有两个规则：**S--->aSQ**和**S--->abc**。第二个规则要求句子是从**ab**起始的，但这里不是的。这样就只剩**S--->aSQ**：



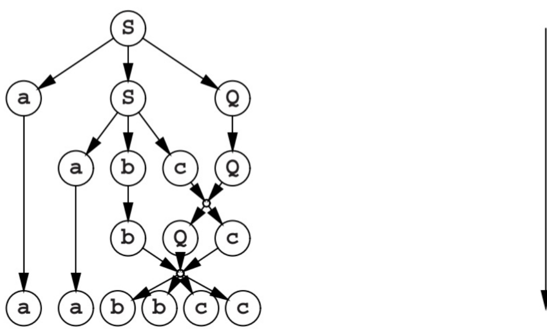
这为我们带来一个很好的解释，为句子中的第一个**a**。现在又有两个规则：**S--->aSQ**和**S--->abc**。一些反射将会揭示在这里第一个规则会是一个坏的选择：**S**的所有生成规则都起始于**a**，并且如果我们能推进到阶段**aaSQQ**，下一步就不可避免的导致**aaa...**，这与输入相矛盾。然而第二个规则，也不是没有问题的：



现在句子起始于**aabc...**，这也与输入相矛盾。然而，有另一条出路：**cQ--->Qc**：



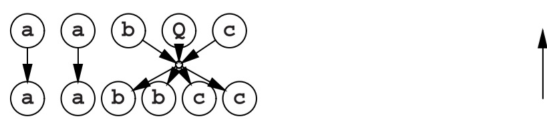
现在就只有一条规则适用了：**bQc--->bbcc**，这样就获得了我们输入的句子（与解析树一起）：



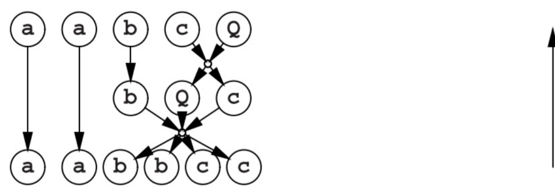
自顶向下解析用前缀顺寻标识了生成规则（并因此特征化了解析树）。

3.2.2 自底向上解析

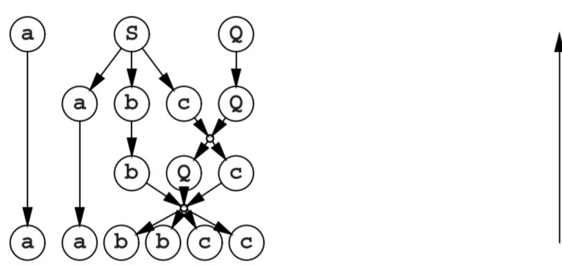
使用自底向上技术，我们如下进行。一个生成步骤必须在最后，并且其结果在字符串中必须是可见的。我们在**aabbcc**中识别出**bQc--->bbcc**的右侧。这给了我们生成的最后一步（也是减少的第一步）：



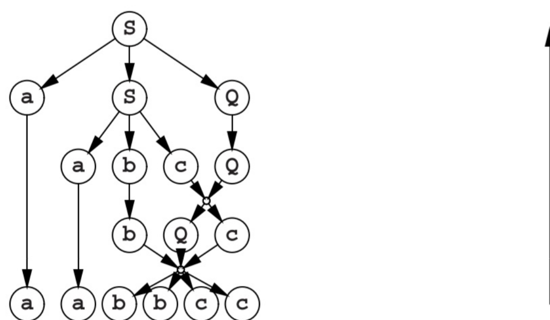
现在我们识别出由**cQ--->Qc**派生的**Qc**：



再次，我们只发现了右侧：**abc**：



以及我们减少的最后一步也让我们没有选择：



自底向上解析倾向于找出前缀顺序中的生成规则。

有趣的是，自底向上解析把解析过程变成了一个生成过程。上述慢慢减少的过程可以被视为用反向语法的生成过程：

aSQ	→	S
abc	→	S
bbcc	→	bQc
Qc	→	cQ

增加一条规则，将起始符号变成一个新的终结符：

S--->!

以及增加一条引入新起始符号的规则，原句是：

I_S ---> aabbcc

如果，从**I**起始，我们能生成我们已经在输入字符串中识别的**!**，并且如果我们记录了我们做过的事情，那我们也就获得了解析树。

生成和减少的双重性被Deussen[21]使用，作为正式语言的一个非常根本的途径。

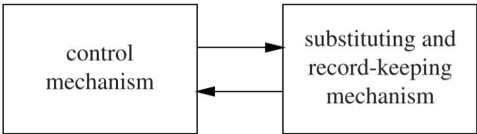
3.2.3 适用性

上述例子表明，自顶向下和自底向上两种方法都在某些情况下能起作用，但同时，也会涉及到一些很微妙的需要注意的事情，这些是我们无法教给一台计算机的。解析文学的几乎整个主体都与正式确定这些微妙的注意事项相关，并且取得了相当大的成功。

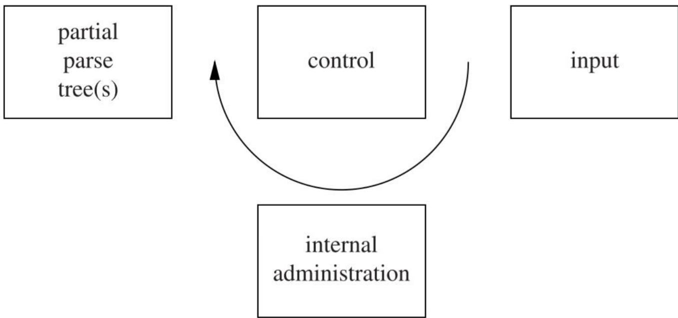
3.3 非确定型自动机

上述两个例子都表征了两个组成部分：一个可以进行替换和记录解析树的机器，以及一个决定机器该做那些动作的控制机器。这个机器相对简单，因为其替换仅允许语法所允许的那些，但是控制机制可以是任意复杂的，并且可能包含语法的广泛的知识。

这种结构可以在所有解析方法中看出：总是有一个替换和保持记录的机器，以及一个指导性的控制机器：



替换的机器被称为非确定型自动机或NDA；被称为“非确定型”是因为经常有几个可能的行动，并且特殊选择没有预先决定，而“自动”是因为会自动执行应激的操作。它管理三个组件：输入字符串（实际上是它的副本），部分解析树和一些内部的管理工作。NDA的每一个行动都转换来自输入字符串的一些信息到部分解析树中，通过管理工作。三个组件可能在过程中进行修改：



NDA的强大力量，以及它实用性的主要来源，就是它可以被轻松的构建，以便它能只做“正确”的行为，也就是说，保持系统部分处理输入，内部控制和部分解析树一致的行为。这有可能导致我们按照选择的任何方式移动NDA的结局：它可能在圈中运动，甚至它可能困住，但是如果它能给我们一个答案，以一个完成的解析树的形式，那这个答案就将是正确的。NDA可以做所有正确的行为也是有必要的，以便它能生成所有解析，如果控制机制足够聪明能指导NDA。NDA的这个属性也很容易安排。

NDA固有的正确性给控制机制带来了极大的自由，简称“控制”。它可能是幼稚的又或成熟的，它可能是麻烦的又或是有效的，它甚至可能是错的，但它绝不可能会使NDA生成一个不正确的解析；而这是一个令人欣慰的想法。然而，如果它是错误的，那可能会是NDA错失一个正确的鸡西，导致无限循环，或在不应该的地方被卡住。

3.3.1 构建NDA

NDA直接源于语法。对于一个自顶向下解析，它主要由语法的生成规则组成以及内部控制是最初的起始符号。该控制机制移动机器直到内部管理与输入字符串相等；接下来解析就被找到了。对一个自底向上的解析器，运动主要由语法的生成规则的反向组成（见3.2.2节），并且内部管理最初是输入字符串。该控制机制移动机器直到内部管理与其实符号相等；接下来解析就被找到了。一个左角解析器像一个自顶向下解析器一样工作，其中一套精心选择的生成规则集合被反转，并且其中有特殊的行为来撤销这个反转当需要的时候。

3.3.2 构建控制机制

构建一个解析器的控制机制是一件相当不同的事情。一些控制机制是独立于语法的，一些定期从语法中查询，一些使用语法预计算的巨大的表格，还有一些甚至使用输入字符串计算的表格。我们应该看到这些的单独的例子：这部分开头就表明的“手动控制”，属于“定期从语法中查询”的类别，回溯解析器通常使用语法独立控制，LL和LR解析器使用语法派生的预计算的表格，CYK解析器使用输入字符串派生的表格，以及Early和GLR解析器使用少数的几个由语法和输入字符串派生的表格。

从语法中构建控制机制，包括表，几乎都是由程序完成的。这种程序被称为解析器生成器；这足够了，语法和也许是终结符号的描述，并且生成一个程序就是解析器。解析器通常由一个驱动和一个或多个表组成，这种情况下被称为表驱动。其中的表可以是具有相当规模并且及其复杂的。

由输入字符串派生的表必须被解析器的一个例行程序计算。应该指出的是，这反映了典型配置，大量不同的输入字符串被根据同一个相对静止不变的语法解析。反面例子是很容易想象的：许多语法被用来试图解释一个给定的输入字符串，例如事件的一个观测序列。

3.4 0型到4型语法的识别和解析

根据一个语法解析句子，原则上我们事先就知道这个字符串实际上源于这个语法。如果我们不能想到更好的东西，我们可以对语法按照2.4.1节的一般生成过程来运行，然后就坐等这个句子出现（我们知道一定会出现）。这情况本身还不足够：我们必须在扩展一下生成过程，以便每个句子形式都能携带其本身一部分的生产树，即便需要在必要的时刻进行更新，但很明显可以通过一点编程工作来完成。我们可能需要等待一会儿（几百万年也说不好）直到句子显示出来，不过最后我们肯定可以收获到解析树。当然这是完全不切实际的，但它至少展示了再理论上，任何字符串都能被解析，如果我们知道它是可解析的，不管是什么语法类型。

3.4.1 时间要求

当解析一个包含几个符号的字符串时，对于解析器的时间要求有些想法是很重要的，即解析输入字符串的一些符号的依赖项所需要的时间。预期的输入长度范围从上千万（自然语言种的句子）到数万（大型计算机程序）之间；一些输入字符串的长度甚至可能是几乎无限长的（一台咖啡售货机再其使用寿命之间的按钮按动序列）。输入长度的依赖项的时间要求也称为时间复杂度。

有几个特征属性让时间依赖项能被识别。一个时间依赖项是指数分布的，如果接下来的每一个输入符号所需的时间都乘以一个常量因子，也就是：每一个额外的输入符号需要双倍的解析时间。指数分布的时间依赖项被写作$O(C^n)$，其中C是增加的常量因子。指数依赖关系，来自于著名的棋盘上每一个各自上的谷物数量都是前一个格子的两倍这个故事；这种方式意味着破产。

一个时间依赖项是线性分布的，如果接下来的每一个输入符号需要花费的处理时间有一个固定的增量；输入长度增加一倍则处理时间增加一倍。这种行为是我们希望在解析器中看到的；解析所需要的时间与读取输入所需的时间成正比。所谓的实时解析器表现的甚至更好：它们可以在读取完成最后一个符号的恒定的时间内生成解析树；如果计算机运行足够快速，它们能跟上以恒定速率输入的无限输入流。请注意，这不一定是真正的线性时间解析器；原则上它们可以读取有n个符号的完整字符串，然后花费与n成正比的时间来生成解析树。

线性时间依赖被写作$O(n)$。一个时间依赖项被称为二次曲线式的，如果处理时间与输入长度的平方成正比（写作$O(n^2)$），以及立方式的如果与长度的三次方成正比（写作$O(n^3)$）。总之，一个依赖项与n的任意次幂成正比则被称为多项式（写作$O(n^p)$）。

3.4.2 0型和1型语法

一个任意的0型语法的识别问题是不可解的，这在形式语言学上是一个显著的成果。这意味着不会有一种算法，它接受一个任意的0型语法以及一个任意的字符串，然后在有限的时间内告诉我们这个语法是否能能生成这个字符串。可以证明这种说法，但是证明过程非常令人生畏，并且它不能提供任何能窥探到造成这种现象的原因。这是一个逆向的证明：我们能证明，如果存在这样一种算法，我们就能构造第二个算法，其能证明语法只能在永远不终止的情况下才能终止。由于这在逻辑上是不成立的，并且由于其余所有的前提在证明过程中都表明我们不得不得出一个结论就是，在我们最初的前提下，0型语法的识别器在逻辑上是不可能的。令人信服，但精神上却难以接受。完整的证明见Hopcroft and Ullman [391, pp. 182-183], or Révész [394, p. 98]。

为一定数量的0型语法构建一个识别器却是很有可能的，通过使用某种特定的技术。然而这种技术却不能对所有0型语法起效。事实上，不论我们收集了多少种技术，总会有这些技术不起作用的语法存在。在某种意义上，我们只是不能使我们的识别器足够复杂。

对于1型语法，这种情况却是完全不同的。1型语法的生成规则不能使句子形式收敛这个看似无关紧要的属性，让我们可以为自底向上的NDA构建一个至少原则上能起作用的控制机制，而不考虑语法。这个控件的内部管理包含一组可能在输入的句子中发挥作用的句子形式；它开始只包含输入的句子。NDA的每一次移动都是一次还原，根据语法。现在这个控制将NDA的所有可能的移动都应用在内部管理的所有句子形式中，以任意顺序的方式，并将每一个结果添加至内部管理中，如果不是已经存在的话。这个动作会一直持续到所有的句子都完成了所有可能的移动并得到了结果。由于没有哪个NDA移动可以使句子形式变长（因为所有的右侧至少是与其左侧的长度相当），并且由于句子形式的数量有限，与输入字符串长短相当，这最终都是会发生的。现在我们在内部管理的句子形式中找到仅只包含起始符号的那一个。如果它确实存在，那我们就识别出了输入字符串；如果不存在，那输入字符串就不属于这个语法所代表的语言。如果我们还记

得，在一些额外的管理机制中，我们是如何得到这个起始符号的句子形式，我们就已经得到了解析过程。而所有的这些都依赖于大量的书记工作，而我们并不打算讨论这个，因为反正也没有人这么做。

综上所述，我们并不能总是为0型语法构建解析器，但如果是1型语法那我们就能做到。为这种类型的语法构建一个实用、合理高效的解析器，是一个非常困难的问题，但是在过去的40年中一直保持缓慢但稳定的进展（见18.1.1节（电子版））。这不是一个热搜话题，主要是因为0型和1型语法是出了名的不友好并且永远不会被广泛应用的。然而并不是完全没有用处，因为一个好的0型语法解析器可能会让定理证明器[1]有一个好的开端。

对人类不友好的思虑并不适用于两级语法。有一个实用的两级语法解析器将棒极了，因为这将会让解析技术（以及其所有的内置自动化技术）应用到更广泛领域中相比于现在，尤其是上下文条件很重要的情况。两级语法解析情况的目前的可能性在15.2.3节中讲述。

所有已知的0型、1型以及无限制的两级语法的的解析算法，都有指数级分布的时间依赖项。

> 1. 一个理论证明程序就是，给定一组定理或公理，在没有或极少认为干预的情况下来证明或反证这个理论的程序。↩

3.4.3 2型语法

幸运的是，CF（2型）语法的解析算法相比于0型和1型要著名的多。使用CF和FS语法中几乎所有实用的解析都做过了，并且在上下午无关语法的解析中几乎所有遇到的问题都被解决了。这么大差距的原因可以在CF语法生成过程中找到：句子形式中的一个非终结符的演变是完全独立于其余非终结符的演变的，并且相反的是，在解析过程中，我们可以结合部分解析树而不理会它们的历史。在上下午相关语法中这些都不是正确的。

自顶向下和自底向上解析过程都很容易适用于CF语法。在下面的例子中，我们应使用简单的语法

```
Sentence_S    →  Subject Verb Object
Subject       →  the Noun | a Noun | ProperName
Object        →  the Noun | a Noun | ProperName
Verb          →  bit | chased
Noun          →  cat | dog
ProperName    →  ···
```

3.4.3.1 自顶向下CF语法解析

在自顶向下CF语法解析中，我们从起始符号入手，并尝试产生输出。这里的关键词是预测和匹配。句子形式中任何时候都有一个最左侧非终结符*A*，而解析器系统的尝试预测一个*A*的恰当的替代品，在这个位置的输入中一旦有兼容的符号被发现，那*A*的产物就应该开始了。这个最左侧的非终结符也被称为预测过程的目标。

想想图Fig 3.5中的例子，其中**Object**就是最左侧的非终结符，也就是“目标”。在这种情况下，解析器将首先预测**Object**的**the Noun**，然后会立即推翻这个替换项，因为**the**的位置被要求是一个**a**。接下来，解析器将会尝试**a Noun**，这个将会被暂时接受。**a**匹配上了，新的最左侧非终结符就成了**Noun**。当**Noun**最终生成了**dog**时，这个解析器就成功了。解析器将会

为**Object**尝试第三次预测，**ProperName**；这个替换项不会被立即拒绝，因为解析器无法知道**ProperName**不能起始于一个**a**。它将在稍后阶段中失败。

Input:	the	cat	bit	a dog
Sentential form:	the	cat	bit	Object
(the internal ad|ministration)				

Fig. 3.5. Top-down parsing as the imitation of the production process

这种方法有两个严重的问题。虽然理论上，它可以处理任意的CF语法，但如果被单纯的照章办事的执行的话，可能会在一些语法中走到死循环中。这个可以通过使用一些特殊技术来避免掉，这在大多数自顶向下解析器中处理了；这个将在第6章中详细讲解。第二个问题是算法必须符合时间指数分布，因为任何一次预测都可能被证明是错误的，并且需要在反复试错中纠正。上面的例子显示了可以通过预处理语法来增加一些效率：其优势在于可以预先知道哪些记号可以开启**ProperName**，以避免预测注定会失败的替代项。这对于语法中的大多数非终结符来说是正确的，而且这种信息可以很容易的从语法中计算出来并存储在一个表中供解析中使用。对于一个合理的语法集合，可以实现一个线性时间依赖项，比如第8章将会解释的。

3.4.3.2 自底向上CF语法解析

在自底向上的CF语法解析中，我们从输入开始，然后尽量减少它与起始符号的差距。这里的关键字是转移和缩减。当我们在过程中时，我们手中有了一个由输入缩减而来的句子形式。在这个句子中的某个地方必定有一个段（子字符串），其是这个句子形式的最后一步生成步骤的结果。这一段对应于生成规则$A \rightarrow \alpha$的右侧α，而且必须被缩减到A。段和生成规则一起被称为句子形式的句柄，有一个相当拟合的表达式；见图Fig 3.6。（当生成规则在段的发现中是显而易见的，那匹配的段通常单独被称为“句柄”。通常我们会遵守这种习惯，不过我们认为称之为句柄段要更明确一些。）

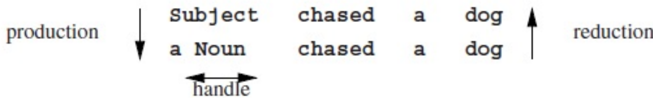


Fig. 3.6. Bottom-up parsing as the inversion of the production process

诀窍是找到句柄。它必须是一个规则的右侧，所以我们从寻找这种右侧开始，通过将句子形式中的符号转换到内部管理机制中。当我们找到这样一个右侧，我们向着其左侧对其进行缩减，然后重复这个过程直到只剩下起始符号。用这种方式我们不一定能找到正确的句柄；如果我们犯错，那将会在接下来的步骤中卡主，然后将不得不撤销一些步骤，然后转向更多的符号并再次尝试。在图Fig 3.6中，我们本可以让**a Noun**向着**Object**缩减，但最后大胆走向了死胡同。

这种方式有两个本质上相同的问题，在自顶向下技术中。它可能产生循环，并且在有ε规则的语法中也可能会这样：它将在所有的地方持续找到空的生成。这个可以通过触及语法来纠正。并且可以花费服从指数分布的时间，因为句柄的正确识别必须通过试错来完成。同样的，对语法做预处理往往会有帮助：在语法中很明显，**Subject**可以通过追逐来找到，但**Object**却不能。所以如果下一个符号是追逐的，那将一个句柄缩减至**Object**是无意义的。

3.4.4 3型语法

在正则语法中，一个右侧包含最多一个非终结符，所以最左边和最右边推到没有区别。自顶向下方式对右正则语法来说有效的多；而对左正则语法来说，自底向上方式要好的多。当我们采取图Fig 2.15中的生成树，并且把它逆时针旋转45。，我们就得到了图Fig 3.7的生成链。非终结符的序列向右滚动，当它们离开的时候就生成了终结符。在解析中，给我们的是终结符而希望得到的是非终结符序列。第一个起始符号是给出的（因此更适用自顶向下）。如果只有一条规则以输入的第一个符号开始，那我们就是幸运的并且知道接下来该怎么做。然后大多数时候，有许多规则是以同一个符号开始的，那时候我们就需要更多的智慧了。至于2型语法，我们当然可以通过试错找到正确的下一步，但存在着更多可以处理任何正则语法的更有效的方式。由于他们中的一些是以更先进的解析技术为基础的，所以我们将在第5章中一一介绍。

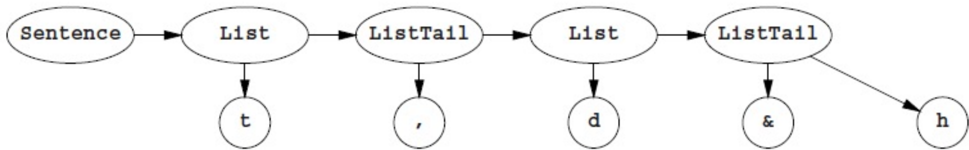


Fig. 3.7. The production tree of Figure 2.15 as a production chain

3.4.5 4型语法

有限选择语法（FC）不包含生成树，并且对一个FC语法语言的一个给定输入字符串的成员关系可以由简单的查找来决定。这种查找方式通常不认为是“解析”，但仍然在这里提及有两个原因。第一，它可以受益于解析技术，第二，在解析环境中通常需要它。自然语言中有一些单词类别，只包含数量非常有限的成员；例如代词、介词和连词。通常快速的决定一个给定单词是否属于这些有限选择类别之一或者是需要在进一步的分析是至关重要的。这同样适用于编程语言中的保留字段。

一种方法是将FC语言作为正则语法来考虑，并应用第5章的技术。这通常是及其有效的。

另一个经常使用的方法是使用一个哈希表。见任何一本关于算法的书，例如Cormen et al. [415], 或者 Goodrich and Tamassia [416]。

3.5 上下文无关解析方法的概述

在Chomsky语法类型中，上下文无关（2型）语法占据着最突出的位置。这有三个原因：1.CF解析的结果在生成树上，这让语义的表达和结合更容易；2.CF语言覆盖了很大一部分人们想要自动处理的语言；3.有效的CF解析是有可能的----虽然有时候存在很大的困难。在重要性上上下文无关语法后紧跟着有限状态语法。这是因为世界和设备是有限的；自动售货机、远程控制、病毒探测器，所有这些都展示了有限状态的本性。本书的其余章节，因此将主要关注CF解析，除了第5章（有限状态语法）之外，以及第15章（非Chomsky体系）。我们现在先看一下上下文无关解析方法的概述。

关于解析文学的读者面临着大量相互之间关系往往不明确的技术。然而现在所有的技术都可以被放在一个单一的框架中，根据一些简单的标准：见图Fig 3.11。

我们已经看到，一个解析技术要么是自顶向下，从起始符号开始重新生成输入字符串，要么是自底向上，向着起始符号缩减输入字符串。下一个分歧点是在定向和非定向解析方法之间。

3.5.1 方向性

非定向性方法构建解析树，当以任何他们认为合适的顺序访问输入字符串时。当然这要求在解析之前，完整的输入字符串要存在内存之中。分别有自顶向下和自底向上两个版本。定向解析器会按照顺序一个接一个的访问输入字符串，持续访问直到更新完部分解析树。也有自顶向下和自底向上两个版本。

3.5.1.1 非定向方法

非定向性的自顶向下方法即简单又直接，可能已经被很多人独立发明出来了。据我们所知最早是Unger[12]在1968年提出的，但是在他的文章中的描述似乎这个方法当时已经存在了。这个方法在文献中一直不受重视，但却比人们所认为的要重要的多，因为它以不知名的方式被大量的解析器使用。我们应该称之为Unger方法；见4.1节。

非定向性的自底向上方法也被很多人独立发现，其中有Cocke (in Hays [3, Sect. 17.3.1]), Younger [10],和Kasami [13]；更早是Sakai [5]提出的。它是以三个最著名的发明家合称命名的CYK（或者说又是称为CYK）。它受到广泛的注意，因为其天生的执行性要比Unger方法更有效率。不过这两种方法的效率都还可以提升，达到大致相同的效果；见Sheil [20]。CYK方法将在4.2节中讲述。

非定向性方法通常首先构建一个数据结构，总结输入句子的语法结构。在第二阶段的时候，解析树就可以从这种数据结构中产生。

3.5.1.2 定向方法

定向方法一个符号一个符号的处理输入字符串，从左到右。（从右到左解析也是可能的，通过使用语法的镜像；这只是偶尔会有用）这样做的好处是，解析可以启动并且确实进行，在输入字符串的最后一个字符出现之

前。定向方法全部都显示或隐式的基于解析自动机上，在3.4.3节中描述的，其中自顶向下方法执行预测和匹配，自底向上方法执行转换和缩减。

定向方法通常可以构建（部分）解析树，当它在处理输入字符串时，除非语法不明确或者需要一些后续处理。

3.5.2 搜索技术

第三种分类解析技术的方式涉及到在指导解析自动机通过其所有可能性找到一个或所有解析结果的搜索过程。

总的来说有两个方法来解决问题，其中有几个备选方案其核心都是：深度优先搜索和广度优先搜索。

- 在深度优先搜索中，我们的注意力集中在一个解决一半的问题上。如果在一个给定点***P***上出现分叉这样的问题，我们先将其中一条放在一边等待稍后处理并继续处理另一条。如果这条路最后失败了（或者即便是成功了，但是我们想要的是全部的结果），我们就回到点***P***在继续处理刚才被放在一边的那条。这就是所谓的回溯。

- 在广度优先搜索中，我们保留了一套解决一半问题的集合。从这个集合中我们计算出一个新的（更好的）解决一般问题的集合，通过检查每个已有的解决了一半的问题；对每一个选项，我们在新集合中创建一个副本。实际上，这个集合最终会包含所有的解决方案。

深度优先搜索的优点是它需要与已存在的问题数量成正比的内存空间，而不像广度优先搜索，它需要的内存是程指数增长的。广度优先搜索的优点是它将首先找到最简单的方案。这两种方法都需要在原则上服从指数分布的时间。如果我们想要更有效的（并且不接受指数分布要求），我们需要一些手段来限制搜索。搜索技术的更多信息，可以在任意一本关于算法的书上找到，例如Sedgewick [417] or Goodrich and Tamassia [416]。

这些搜索技术根本不限于解析技术，而是可以在广泛的范围内使用。一种传统的用法是找到迷宫的出口。图Fig 3.8（a）是一个简单的有一个入口和两个出口的迷宫，Fig 3.8（b）描绘了深度优先搜索将会采取的路径；这对行走其中的人类来说是唯一的选：他不能复制他自己获取迷宫。死胡同让深度优先搜索回溯到最近的一条没有尝试过的选择。如果搜索者也回溯每一个出口，那他将发现所有的出口。Fig 3.8（c）显示了广度优先搜索在每

个阶段已经检查过的空间。死胡同（阶段3）将导致搜索分之被丢弃。广度优先搜索将会找到最快的捷径（最短解决方案）。如果它持续下去直到没有分支剩下，那也将找到全部的出口（所有解决方案）。

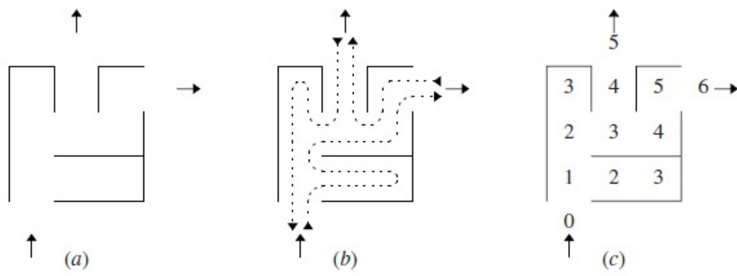


Fig. 3.8. A simple maze with depth-first and breadth-first visits

3.5.3 一般定向方法

解析就是重建生成过程这个观点，当使用定向方法时就尤为明显。它概述在以下两点中：

- 一个定向的自顶向下（从左到右）的CF解析器，标识生成过程中最左侧的产物。

并且

- 一个定向的自底向上（从左到右）的CF解析器，标识反向生成过程中最右侧的产物。

我们用一个非常简单的语法来证明这个：

$$
\begin{array}{lcl}
S_S & \rightarrow & P\ Q\ R \\
P & \rightarrow & p \\
Q & \rightarrow & q \\
R & \rightarrow & r
\end{array}
$$

这个语法只生成唯一的字符串， **pqr** 。

pqr 最左侧的产物的生成过程如下：

```
   |S
1  |P  Q  R
2  p  |Q  R
3  p  q  |R
4  p  q  r |
```

其中|标识生成过程进行到哪儿了。自顶向下过程通过首先确定生成 **p** 的规则来模仿这一过程，**P--->p** ，接下来是 **q** ，等等：

$$
|S \overset{S\rightarrow PQR}{\underset{(1)}{\Rightarrow}} |PQR \overset{P\rightarrow p}{\underset{(2)}{\Rightarrow}} p|QR \overset{Q\rightarrow q}{\underset{(3)}{\Rightarrow}} pq|R \overset{R\rightarrow r}{\underset{(4)}{\Rightarrow}} pqr|
$$

pqr 的最右侧的生成过程如下：

```
  S|
1 P Q R|
2 P Q| r
3 P| q r
4 | p q r
```

同样|标识生成过程进行到哪儿了。自底向上分析回滚了这一过程。要做到这一点，必须要先确定步骤4的规则，**P--->p**，并将它作为还原，然后是第3步，**Q--->q**，等等。幸运的是解析器可以很容易做到这点，因为最右侧的生成产物成为了已生成和未生成的句子部分的分界，所以最后的产物是结果的最左侧，正如我们前边看到的。然后解析过程就可以从那儿开始了：

$$|pqr \underset{(4)}{\overset{P\to p}{\Rightarrow}} P|qr \underset{(3)}{\overset{Q\to q}{\Rightarrow}} PQ|r \underset{(2)}{\overset{R\to r}{\Rightarrow}} pqr| \underset{(4)}{\overset{S\to pqr}{\Rightarrow}} S|$$

这种双重逆转是定向自底向上解析所固有的。

解析树的构建和句子形式的关联如图Fig 3.9所示，其中虚线表示句子形式。左侧我们有一个完整的解析树；对应的句子形式就是终结符的字符串。中间的图展示了在一个自顶向下解析器中的部分解析树，在 **p** 生成之后。对应这种情况的句子形式就是 **pQR** 。它起因于两种生成过程 **S** □ **PQR** □ **pQR**，这产生了解析树。右图表示了在自底向上解析器中 **p** 被生成后的部分解析树。对应的句子形式是 **Pqr** ，由 **pqr** □ **Pqr** 产生；唯一的还原结果导致了只有一个节点的解析树。

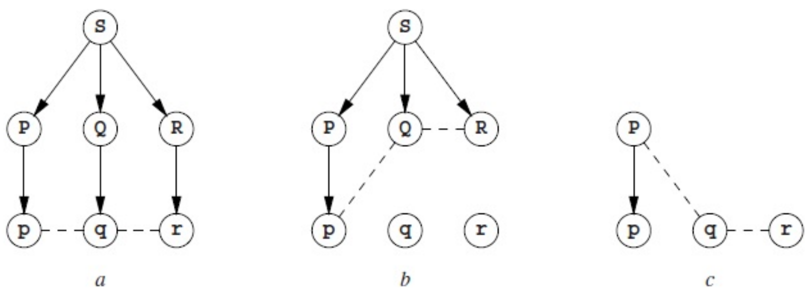


Fig. 3.9. Sentential forms in full parse tree (*a*), during top-down (*b*), and during bottom-up (*c*)

将广度优先或深度优先与自顶向下或自底向上结合起来，就得到了四种级别的解析技术。自顶向下技术将在第6章介绍。深度优先自顶向下技术带来了一个非常简单的实现，叫做递归降序；这种技术将在6.6节介绍，非常适

合手工编写解析器。由于深度优先搜索内置于Prolog语言中，对这种语言的大量语法的递归降序解析就可以非常简单的定制，使用一个被称为“Definite Clause Grammars”（6.7节）。这项技术的适用性可以扩展到所有语法通过使用一种叫做“cancellation”(6.8 节) 装置。

自底向上技术将在第7章介绍。广度优先和自底向上的结合带来了Earley解析器，对CF语法来说其中有一些非常有效和流行的解析器（7.2节）。一个正式的相似但执行却非常不同的方法带来了“图标解析”（7.3节）。

Sudkamp [397, Chapter 4]给出了一个完全正式的解释，关于[广度优先|深度优先][自顶向下|自底向上]上下文无关解析。

3.5.4 线性化方法

上一节所示的大多数一般搜索方法，最坏的情况都依赖于时间指数分布：输入的每个附加符号都将解析时间乘以一个常数因子。这些方法只在输入长度非常小的情况下适用，大约20个字符就是最大值了。即便是上述方法中最优的，在最坏的情况下也要求立方次时间：10个令牌时需要1000次操作，100个令牌时需要1000000次操作，而10000个令牌（一个较大的计算机程序文件）时则需要10^{12}次操作，即便每次操作只需10纳秒也要花费至少3小时的时间。显然在真实的速度下，我们更希望有一个线性时间的计算方法。不幸的是，至今没有找到这样一种方法，尽管没有证据表明这种方法是不存在的，但种种迹象却表明似乎情况就是如此了；详情见3.10节。将此与非严格短语结构解析方法相比，就可以证明没有这种算法存在了（见3.4.2节）。

因此，同时或者永久的，我们将不得不从我们的目标中拿掉一个对象，线性时间通用解析器。我们可以有一个最优也是立方次时间依赖的通用解析器，或者是一个不能适用于所有CF语法的线性解析器，但这两者不能兼有。幸运的是，有延时解析方法（特别是LR解析），可以处理大量语法种类，但依旧如果一个语法只是用最自然的方式来描述一个预期的语言而没有涉及到分析方法，那就只有很小的机会能使用自动线性分析。在实践中，语法通常首先是为了自然而设计，然后通过手动调整来符合现有分析方式的要求。这种调整相对简单，具体取决于所选择的方法。简而言之，对于任意给定语法做一个线性解析器有10%的工作是艰难的，而另外90%可以由计算机来完成。

我们可以通过限制非确定性解析自动机的处理位移数量在每种情况下只有一个来实现线性解析时间。由于这种情况下一个自动机的处理位移没有别的选择，所以被称为“确定性自动机”。

确定性自动机的处理位移是由输入流明确确定的（现在我们可以说是流，因为自动机是从左到右处理的）。这个的序列就是一个自动机能给出的一个句子的唯一一个解析。如果语法是明确的那这就是正确的，单如果语法

不明确，那确定性自动机将会把我们定死在一个特定的解析中。我们将在8.2.5.3节和9.9节中细说。

剩下的就是解释如何从语法中推导出解析自动机的确定性控制机制。由于对这个问题没有一个单独的很好的解决办法，那么存在很多的次优的解决方案就不足为奇了。从一个非常广泛的角度来说，它们都使用同样的技术：它们都深入分析语法，以使可用于识别死角的信息被发现。然后就可以避开这些死角。如果应用于该语法的方法，能够避开足够多的死角并使得不在存在死角，那么这个方法对于语法来说就是成功的，并且给我们提供了一个线性时间解析器。或者它失败了，那我们要么换另一个方法要么就改变语法使之能够使用该方法。

与迷宫问题的一个大致的类比可能使让这点更清晰。如果允许我们在迷宫中做预处理（不太可能但有启发性），那下面的方法会使我们的搜索成为确定性的。我们先假定迷宫由正方形房间的网格构成，如果3.10(a)所示。深度优先搜索将在迷宫中找到一个13次移动的路线（图3.10(b)）。现在我们对迷宫预处理如下：如果有一个房间有3面墙，那么给其加上第4面墙，持续进行直到没有拥有3面墙的房间。如果现在所有的房间只有两面或者四面墙壁，并且没有别的选择，那么我们就成功了；见图3.10(c)，现在路径只有5次移动，并且不需要搜索了。通过这个方法我们就看到如何能将死角识别出来了，用以帮助缩小选择范围。

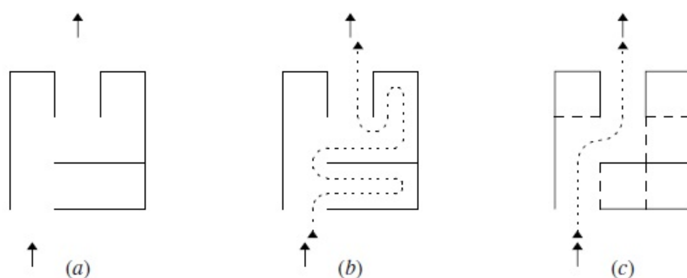


Fig. 3.10. A single-exit maze made deterministic by preprocessing

有一点要说明的是，上述的类比是有限的。它只关心迷宫这一个被预处理的对象。在解析中我们要关注两个对象，一个是静态的并且能被预处理的语法，还有一个是变化中的输入。（但请参阅问题3.6 以扩展类推方法。）

回到解析自动机，我们可以说，它的确定性更精确：一个拥有前瞻符号*K*的解析自动机是确定性的，如果它的控制机制可以，那么给予内部管理和下一个符号*K*输入信息，以明确决定下一步做什么——要么匹配要么预测以及在自顶向下情况中预测什么，要么转移要么减少以及在自底向上情况中如何减少。

它的原因是确定性自动机创建了一个线性时间解析器，但这并不完全明显。解析器可能在有限的时间内知道下一步该做什么，但对于给定输入令牌可能有很多步要执行。更具体的说，某些确定性技术对于给定位置*k*会需要*k*步，这表明二次行为是可能的（见问题3.5）。但是每个解析步骤要么创建一个新的节点（预测或减少），要么消耗一个输入令牌（匹配和转移）。两个操作都只能执行*O(n)*次，其中*n*是输入的长度：第一个是因为解析树的大小仅为*O(n)*，第二个是因为只有*n*个输入令牌。因此不论各种各样的任务的操作是如何分布的，它们的总和不能超过*O(n)*。

与语法类型一样，确定性解析方法也可以用缩写表示，就像LL，LALR等。如果一个方法*X*使用了前瞻符号*k*，那就写作*X(k)*。所有确定性方法都需要以某些方式预处理语法来推导出解析自动机，外加一个解析算法或用自动机来处理输入。

3.5.5 确定性自顶向下和自底向上方法

只有一种确定性自顶向下方法，它叫做*LL*。第一个L代表从左至右（Lift-to-Right），第二个是“确定最左侧生成”（identifying the Leftmost production），就像定向自顶向下分析器一样。LL解析在第8章中进行讲解。LL解析，尤其是LL(1)非常受欢迎。LL(1)解析器通常是由一个解析器生成器生成的，但是一个简版的可以通过手工费点劲写出来，通过递归减少技术；见8.2.6节。偶尔将LL(1)方法从输入的最后一个令牌开始，这时就叫做*RR(1)*。

有很多的确定性自底向上方法，最强大的叫做*LR*，其中L也是值从左至右（Lift-to-Right），R代表“确定最右侧的生成”（identifyingthe Rightmost production）。线性自底向上方法将会在第9章讲到。他们的解析器只能是由一个解析器生成器生成的：这样一个解析器的控制机制太过复杂以至于人工是无法完成的。部分确定性自底向上方式非常的受欢迎，甚至可能比LL(1)方式更广泛的使用。

LR(1)解析比LL(1)解析更强大，但也更难以理解和不那么方便。其他方法无法随便同LL(1)相比。17.1节有实用解析方式之间的对比。LR(1)解析也能从输入最后端开始，此时被称作*RL(1)*。

这两种技术都使用了前瞻性来确定下面的行动。通常这种前瞻性会被一个令牌的限定（LL(1)，LR(1)等），或者至多少数的几个令牌的限定，但偶尔不受限的前瞻能带来很大帮助。这需要其他的解析技术，而这会带来确定性解析器的细化分类，见图3.11。

自顶向下和自底向上方法的差异很大，当我们细看不同分析器的选择时就很容易理解了。自顶向下解析器本质上没有什么选择：如果一个终结符已经被预测，那就没有别的选择而只能从存在的匹配中确定；只有当一个非终结符被预测到了，它才有一个选择那个非终结符的选择。一个自底向上解析器移向下一个输入符号，即便是缩减的也是可以的（并且也必须这样

做）。此外，如果可能存在缩减，它还可能会选择在右侧的一组中进行选择。总的来说，它比自顶向下解析器有更多的选择，所以需要更强大的技术来确定它。

3.5.6 非规范方法

对于许多实用语法来说，上述方法依旧不能产生一个线性时间确定性解析器。通常采取的方式是对语法稍作修改已适用于选择的方法。但不幸的是，最终的解析树无法对应于原始的语法，后续还需要在手动修改一次。另一种方法是设计一个解析器，让其一直推导直到没有可用的信息，然后以“半猜测”的方式持续解析直到信息再次可用。这样的解析器被称为非规范的，因为它们以非标准的方式在识别解析树中的节点，“非规范”顺序。不用说，这肯定得谨慎的实施，一些最强大、最智能、最复杂的确定性解析算法就属于这个范围内。第10章中会讲到。

3.5.7 广义线性方法

当我们构建一个确定性控制机制的试图失败，并留给我们非确定性的但是又几乎是确定性的机制时，我们还不需要感到绝望：我们可以回到广度优先搜索，在试验期间内解决留下的非确定性问题。我们原本的方法越好，遗留的不确定性就越少，需要用到广度优先搜索的就越少，那解析器的效率就越高。这种解析器被称为“广义解析器”；对大多数自底向上和自顶向下确定性方法，广义解析器已经设计出来了。在第11章中有介绍。广义LR（或GLR）（Tomita [162]）是现今可用的最佳通用CF解析器之一。

当然，重新引入广度优先搜索是我们的一个冒险。语法和输入有可能会使每个输入中隐藏不确定性，这样会导致我们的解析器再次具有时间依赖性。然而在实践中，从来没有发生过这种情况，这样的解析器非常有用。

3.5.8 总结

图3.11总结了本书中涉及的解析技术。Nijholt [154]绘制了一个更抽象的分析视图，基于左角解析。更抽象的一个概要参见Deussen [22]。Griffiths和Petrick [9]在早期做了一个系统的调查。

	Top-down	Bottom-up
Non-directional methods	Unger parser	CYK parser
Directional Methods, depth-first or breadth-first	The predict/match automaton recursive descent DCG (Definite Clause Grammars) cancellation parsing	The shift/reduce automaton Breadth-first, top-down restricted (Earley) chart parsing
Deterministic directional methods: breadth-first search, with breadth restricted to 1, bounded look-ahead	LL(k)	precedence bounded-right-context LR(k) LALR(k) SLR(k)
unbounded look-ahead	LL-regular	DR (Discriminating-Reverse) LR-regular LAR(m)
Deterministic with postponed (“non-canonical”) node identification	LC(k) deterministic cancellation Partitioned LL(k)	total precedence NSLR(k) LR(k,∞) Partitioned LR(k)
Generalized deterministic: maximally restricted breadth-first search	Generalized LL Generalized cancellation Generalized LC	Generalized LR

Fig. 3.11. An overview of context-free parsing techniques

3.6 解析技术的力量

一般来说，一个T_1解析技术要比一个T_2解析技术更有力量（更强大），当T_1能够处理T_2所能处理的全部语法，而不是相反的情况。不正式的说法是如果占主动权的一方可以比另一个处理更多的语法时，那这个解析技术就比另一个要强。不过这当然是无稽之谈，由于所有的解析技术都能处理无限集合的语法，那么“更大”的概念就是难以定义的。此外，一个没有明确目的的用户语法几乎没有任何可能能符合现有的解析技术。从用户角度来看，对“平均”实用语法的修改已适用于方法T来处理的工作量才是有意义的，以及修复这种修改对解析树带来的差异。解析技术T的力量（作用）与这个工作量是成反比的。

一个强大的解析器总是更复杂的，相比于弱小的解析器需要花费更多的精力来编写它。但由于一个解析器或者解析器生成器（见17.2节）只需要费一次功夫编写好之后，就可以想怎么用就怎么用，所以长远来说一个强大的解析器更省时省力。

虽然一个“强大”的解析器的概念很直观明了，但当解析器和语法混在一起时就很容易出现混淆了。解析器越强大那么语法的限制就越小，而“弱小的”解析器就要限制语法了。通常会使用语法来命名解析器，而这也就是混乱开始的地方。一个“强大的LL(1)语法”比“LL(1)语法”有更严格的限定；也可以说是更强大的LL(1)。这样的语法的解析器很简单，相比于（完整）LL(1)语法，并且也是属于弱小的--可以承受的。所以实际上，一个强大的LL(1)（strong-LL （1））解析器比一个LL(1)解析器要弱小一些。我们一直都在强调“强大（strong）”和“LL(1)”之间的连字符号，以表明“强大”是用来形容“LL(1)”的，而不是语法，但并非所有的出版物都会遵循这个约定，所以读者一定要清楚这一点。“弱化优先分析器”出现时情况就反过来了，“弱化优先分析器”要比“优先分析器”更强大（尽管还有一些别的不同）。

3.7 解析树的表现形式

解析的目的是获取一个或多个解析树，但很多解析技术不会提前告诉你会有0个、一个、几个或者无限多个解析树将会生成，所以对于即将出现的结果会缺点准备。有两件事情我们要避免：无准备应对和没有解析树，以及准备过渡和分配过多的内存。目前没有太多的文章或书籍涉及到这个问题，但在论文中遇到的技术性问题可以分成两个模型：生产者-消费者模型和数据结构模型。

3.7.1 生产者-消费者模型中的解析树

在生产者-消费者模型中，解析器是生产者，而使用解析树的程序是消费者。正如在计算机科学中的生产者-消费者模型，最直接的问题是，哪个是主进程哪个是子进程。

最好的解决方案是将他们视为同等的，然后可以使用协同例程（*coroutine*）。协同例程在其他的编程语言和编程技术原理中进行了介绍，例如高级程序语言设计，作者R.A. Finkel (Addison-Wesley)。在网络上也有很多的解释。

在协同例程模型中，用户对新解析树的需求和解析器对解析树的提供是由协同例程自动配对的。协同例程的问题是，它们必须被内置到编程语言中，并且没有主流的编程语言适应它们。因此，对解析树表示方法协同例程并不是一个实用的解决方案。

协同例程的现代表现方式，线程，其中配对是由操作系统或编程语言内一个轻量级操作系统来完成的，在一些主要语言中是可用的，但关于并行性概念的介绍并不是解析所固有的。Unix管道有相似的通信属性，单对解析来说却相差更大。

通常解析器是主程序，而消费者是子例程。每当解析器完成解析树的构造时，它就会将树当做参数用一个指针来调用消费者例程。然后消费者就可以决定如何使用解析树：拒绝它、接受它、存储它以备将来的比较，等等。在这个设置中，解析器可以很好的生成解析树，但是消费者可能必须在每次调用之间存储状态数据，以便能够在解析树之间进行选择。这是解析器设计中通常的设置，在这里只有一个解析树，而消费者状态保存则不是一个问题。

还可以将消费者作为主程序，但这会给解析器带来沉重的负担，因为解析器必须在生成解析树的过程中保存全部未完成解析的状态数据。由于堆栈上的状态不能保存为状态数据（除非采用苛刻的方法），这种设置只适用于不使用堆栈的解析方法。

对着这些设置，在大多数情况下用户还有两个问题。首先，当解析器生成多个解析树时，消费者将它们作为独立的解析树接收，然后可能需要做相当大的比较来找出它们之间的差别来进行进一步的决策。第二，如果语法是无限模糊的，并且解析器解析器产生无限多的解析树，那这个过程将不会停止了。

所以生产者-消费者模型对于确定性语法是一个令人满意的方案，但更多的情况下却会有问题。

3.7.2 数据结构模型中的解析树

在数据结构模型中，解析器构造一个独立的表示所有解析树同时进行的数据结构。令人惊讶的是，即使是无限模糊的语法也可以解决；并且，它可以在一个与输入字符长度的3次幂成正比的空间里完成。有人说数据结构有立方结构依赖性。

有两种表现形式：解析林和解析林语法。虽然两者在本质上是相同的，但在概念和实际使用中却是大有不同，将它们视为单独的个体是有必要的。

3.7.3 解析林

由于森林只是树的集合，所以解析林最天然的形式是由一个单一的节点组成，而解析林中的所有树都可以直接访问。图3.2中的两个解析树合并到图3.12中的解析林中，其中节点中的数字引用了图3.1的语法中的规则号。

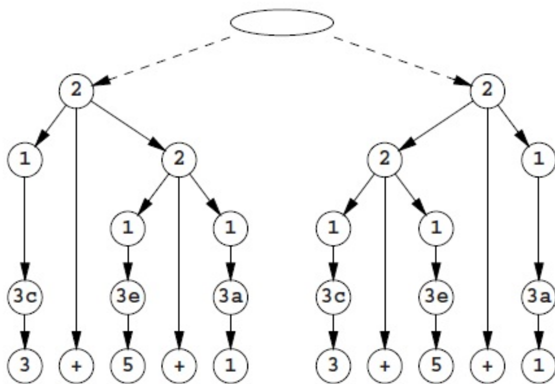


Fig. 3.12. Naive parse forest from the trees in Figure 3.2

当我们看到这幅图时，我们会注意到两件事情：虚线箭头与实线箭头的含义不同；结果树包含大量重复的子树。而且还会好奇顶部的空节点应该是什么。

虚线箭头的含义是“或-或”：顶部的空节点指向左边或是右边的标记2的节点，那么实线箭头就是“与-与”：左边标记2的节点由一个标记Sum的节点与一个标记+的节点与一个标记Sum的节点组成。更特别的是，顶部的空节点，应该被标记为Sum指向规则2的两个应用程序，其中每一个都生成**Sum + Sum**；最左侧的**Sum**指向一个规则1应用程序，第二个**Sum**指向一个规则2应用程序，等等。图3.13展示了完整的与-或树，这里我们看到了一个标记非终结符的节点的替换，即或节点，以及标记了规则号的节点，即与节点。A非终结符的一个或节点对A的终结符的子节点有一个规则号；一个规则号的与节点有右侧规则的子节点的组件。

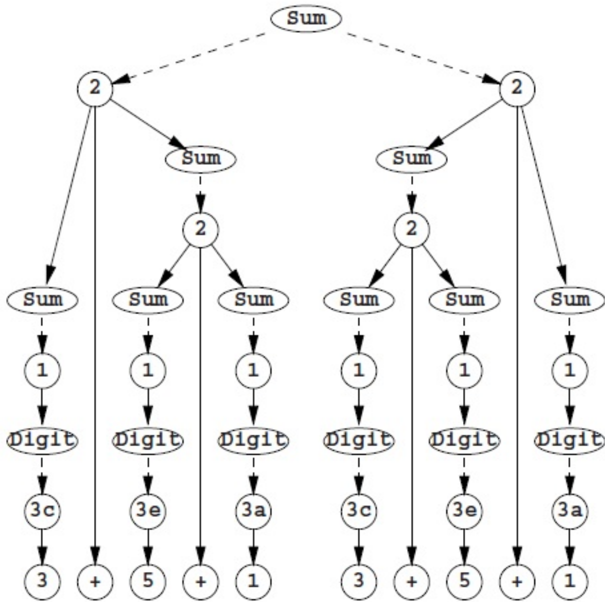


Fig. 3.13. The naive parse forest as an AND-OR tree

3.7.3.1 合并重复子树

现在我们可以将解析林中的重复子树合并在一起了。我们通过保留标记了非终结符A以及跨越输入的给定子字符串的一个副本来做到。如果*A*以多种方式来生成子字符串，那么多个或箭头将从标记了A的或节点出发，每一个都指向一个标记了规则号的与节点。这样，与-或树就变成了一个有向无环图，一个有向无环图，其实正确的叫法应该是一个解析有向无环图，也就是“解析林”变得更加通常了。我们示例的结果如图3.14 所示。

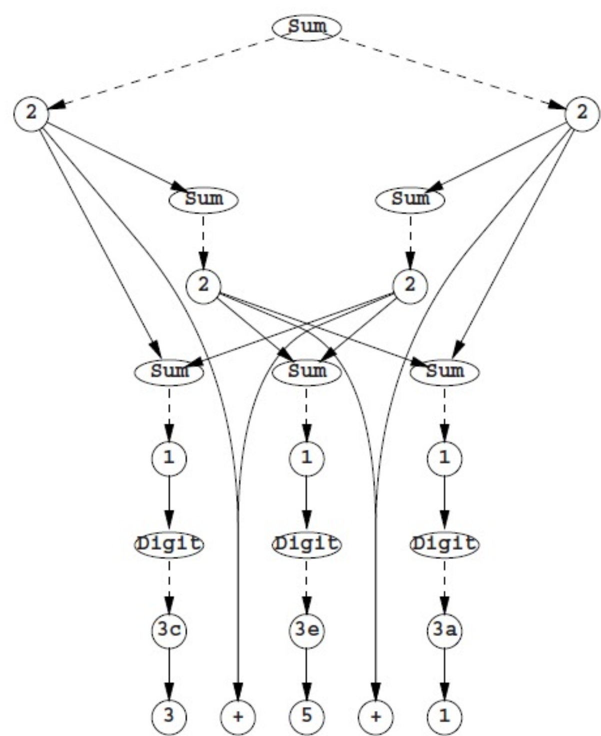


Fig. 3.14. The parse trees of Figure 3.2 as a parse forest

必须注意的是，两个或节点（这代表着规则的右侧）只能在两个节点的其他所有对应节点都相同时才能合并。它不会将图3.14中顶部正下方的两个标记为2两个节点合并；即便它们都是**Sum+Sum**，因为其中**Sum**和**+**都是不一样的。如果将它们合并在一起，那解析林会展示出比对应输入更多的解析树；参见问题3.8.

合并重复子树的过程可以在解析过程中，而不是所有解析树都完成之后。这明显是更高效的，并且有额外的优势，它允许在无限模糊的解析展现在有限的数据结构中。然后解析林就包含循环（环），实际上就是解析图。

图3.15总结了各种Chomsky语法类型相对于歧义的情况。请注意，有限状态和上下午相关语法不能无限模糊，因为它们不能包含可以为空（nullable）的规则。有关生产数据结构的类似摘要，请参见图2.16。

Grammar type	Most complicated data structure unambiguous	ambiguous	infinitely ambiguous
PS	dag	dag	graph
CS	dag	dag	—
CF	tree	dag	graph
FS	list	dag	—

Fig. 3.15. The data structures obtained when parsing with the Chomsky grammar types

3.7.3.2 从解析林中检索解析树

解析林的接收器有多个选择。例如，可以从它生成一个解析树序列, 或者更可能的是, 数据结构可以被修改以剔除各种原因产生的解析树。

从解析林中生成解析树基本上很简单：或-或箭头的每一个选择的组合都是一个解析树。实现应该是从上到下的，并且在这里可以简要的勾勒出来。我们对图做深度优先访问，对于每一个或节点，我们将向外指出的虚线箭头转换为实线箭头；我们将这些选择记录在一个回溯链中。当我们完成了深度优先探索后我们就修复了一个解析树。当我们完成后，检查最近的选择节点，由回溯链提供的最后一个元素，然后如果可以的话做一个与之前不一样的选择；如果不可用那么就后退一步，如此。当我们回溯完整个回溯链，我们就找到了所有的解析树。在图3.16中展示了一个解析树的实现过程。

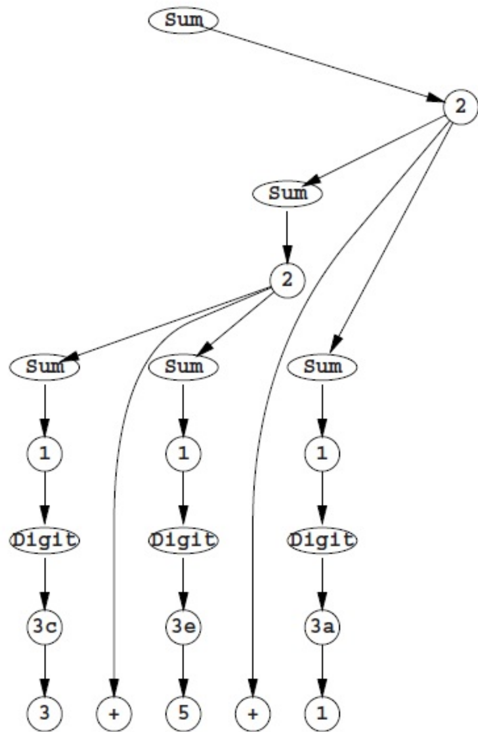


Fig. 3.16. A tree identified in the parse forest of Figure 3.14

首先挑出解析树是很好的选择。如何完成这取决于挑选标准，但常用的技术如下。在解析林中的每个节点都添加了信息，其方式与属性语法的内容类似（2.11.1节）。无论何时当该节点的信息与该节点的类型相冲突时，都将该节点从解析林中移除。这通常会导致其他一些节点成了从上至下无法到达的情况，这时也可以将它们一起移除。

对图3.14的解析林进行有意义的挑选可以基于以下依据，**+**运算符是左关联的，这意思就是**a+b+c**实际是（（**a+b**）**+c**）而不是（**a+**（**b+c**））。然后, 对于每个具有**+**运算符的节点, 其右侧的操作数不能是具有**+**运算符的非终结符。我们看到，图3.14中标记为2的左上节点违反了这个规则：它有一个**+**运算符和一个有一个节点（2）以及一个**+**符号（位置4）的非终结符（Sum）。因此，这个节点可以被移除，以及两个子节点也可以移除。图3.16中的解析书依旧存在。

上面的规则是在算术表达式中关于运算符优先级的一个（非常）特殊的例子；请参阅问题3.10以了解更普遍的情况。

3.7.4 解析林语法

将解析的结果作为语法展示可能看起来很奇怪，甚至是有些令人失望的；毕竟应该从语法和字符串开始，做的所有解析工作，难道最后只是为了成为另一个语法？但是我们将看到解析林有相当多的优点。但这些优点都不明显，这可能也是为什么解析林直到上世纪八十年代才被提出，由Lang [210, 220, 31]引入。而“解析林语法”这个词似乎是被van Noord [221]首次使用。

图3.17 将图3.2 的解析树显示为一个解析林语法，而这之间的过程很有趣。对于原始语法中的每一个非终结符*A*，会从*i*的位置开始产生一个长度为*l*的段，在解析林语法中有一个非终结符*A_i_l*，以及显示*A_i_l*如何产生一个段的规则。例如，解析林语法存在的***Sum_1_5***展示了*Sum*产生了整个的输入字符串（从位置1开始，长度为5）；***Sum_1_5***有多个规则的事实表明, 解析是不明确的；两个规则显示了***Sum_1_5***产生整个输入字符串的两种可能方式。当我们使用这种语法来生成字符串时，它只生成***3+5+1***的输入字符串，但是生成了两次，由于不明确而导致。

```
      Sum_S  -> Sum_1_5
    Sum_1_5  -> Sum_1_1 + Sum_3_3
    Sum_1_5  -> Sum_1_3 + Sum_5_1
    Sum_1_3  -> Sum_1_1 + Sum_3_1
    Sum_3_3  -> Sum_3_1 + Sum_5_1
    Sum_1_1  -> Digit_1_1
  Digit_1_1  -> 3
    Sum_3_1  -> Digit_3_1
  Digit_3_1  -> 5
    Sum_5_1  -> Digit_5_1
  Digit_5_1  -> 1
```

Fig. 3.17. The parse trees of Figure 3.2 as a parse-forest grammar

我们写做*A_i_l*而不是*Ai,l*，因为*A_i_l*代表了一个语法符号的名称，而不是元素*A*的一个下角标；并没有A表或矩阵。*A_i_l*和*A_i_m*之间也没有任何关系；每个*A_i_l*都是一个单独的语法符号名称。

现在说说优势。首先，解析林语法以图形化的方式实现，这一概念已经在上一节中隐晦的表达了：即应该有一个实体来描述一个给定的非终结符如何生成给定的子字符串。

其次，它具有数学美：现在解析一个字符串可以可以看做一个函数，将一个语法映射到一个具体的语法或一个错误值。而不是三个概念——语法、输入字符串和解析林——现在我们只需要两个：语法和输入字符串。更实际的是，所有用于处理原始语法的软件也可以应用到解析林语法。

第三点，解析林语法在梳理过后很容易恢复，使用2.9.5中的算法。例如，将前一节的消歧法则应用于图3.17中的语法规则，可以确定**Sum_1_5**的第一条规则是相违背的。去除这一条规则并应用语法恢复算法生成图3.18的明确的语法，它对应于图3.16中的树。

```
     Sum_S  -->  Sum_1_5
   Sum_1_5  -->  Sum_1_3 + Sum_5_1
   Sum_1_3  -->  Sum_1_1 + Sum_3_1
   Sum_1_1  -->  Digit_1_1
 Digit_1_1  -->  3
   Sum_3_1  -->  Digit_3_1
 Digit_3_1  -->  5
   Sum_5_1  -->  Digit_5_1
 Digit_5_1  -->  1
```

Fig. 3.18. The disambiguated and cleaned-up parse-forest grammar for Figure 3.2

第四，无限模糊解析是没有大的影响的：解析林语法只生成无限多个（相同的）字符串。而产生无穷多的字符串正是语法通常所做的。

最后但同样重要的一点，它很好的适应了解析作为一个新兴而又有希望的方法的交点，这点将在第13章进一步讨论。

现在可以说，解析林和解析林语法实际上是相同的，前者的指针在后者中被称为指代，但这也并不完全一样。指代要比指针更强大，因为指针只能指向一个对象，然而一个指代可以用来识别多个对象；通过重载或不明确指代：指代是多路径的指针。更具体的说，在图3.17中**Sum_1_5**指代了两

个规则，因此在图3.14中承担了顶部或节点的角色。我们看到，在解析林语法中，我们不费丝毫就得到了一个与或树机制，因为它是建立在语法生成的机制上的。

3.8 什么时候才是完成了解析呢？

由于非决定性解析器是一次性处理整个输入字符串，并将其汇总到一个单一的数据结构中，然后可以从中提取出解析树，那么何时能完成解析的问题就不大会出现了。当数据结构完成后，第一阶段就完成了；而提取解析树是在数据结构用完之后或者用户满意就完成了。

原则上，一个定向解析器处于接受状态而所有的输入都已经结束，就是完成了。但这本来是重复要求的，有时其中一个条件就蕴含了另外一个；并且通常其他条件会起同样的作用。因此，对定向解析器来说，这个答案很复杂，取决于很多因素:

- 解析器是否在输入的末尾？就是说，它是不是处理完了输入的最后一个符号？
- 解析器是否处于可接受状态？
- 解析器是否可以继续，例如，如果有下一个字符，解析器是不是能继续处理它？
- 解析器是用来生成解析树的还是只是作为一个识别？第一种可能出现几种不同情况；第二种的话我们的答案只会是是/否。
- 如果我们想要解析数，那是想要全部的解析树还是一个就够了呢？
- 解析器必须要接受整个输入，还是用符合语法的合适的隔离符来进行分割？（如果有一个字符串x，是另一个字符串y的开头部分，那x就是一次分割。）

关于我们是否完成了解析的问题的答案，在下表有了对照，其实EOI代表“输入结束”，yes/no代表对选项的回答。

at end of input?	can continue?	in an accepting state? yes	no
yes	yes	prefix identified / continue	continue
yes	no	OK (yes)	premature EOI (no)
no	yes	prefix identified / continue	continue
no	no	prefix identified & trailing text (no)	error in input (no)

有些答案在直觉上是合理的：如果解析器持续保持在不接受输入的状态下，它应该这样做；如果解析器不能保持在不接受输入的状态，那么输入中存在错误；并且如果解析器在输入结束时仍旧保持可接受状态，那么解析就成功了。但另一些情况要更复杂：如果解析器是在处于可接受状态，我们就会隔离一个前缀，即使解析器在结尾处可以继续“与/或”处理。如果这是我们想要的，那我们可以停止了，但通常情况下，只要可以继续我们就会想要继续：语法**S--->a|ab**以及输入**ab**，我们可以在**a**和声明**a**的一个前缀后结束，但很可能的是我们会想要继续下去，直到**ab**整个被解析结束。这可能是事实，即便我们已经处于结尾处：语法**S--->a|aB**，其中**B**生成**ε**，我们要继续输入**a**以及解析**B**，如果我们想要获取所有的解析。如果解析器做不到，我们在语言中识别了一个字符串，错误信息通常被称为“尾随垃圾”（trailing garbage）。

请注意，“过早处于结尾（premature EOI）”（在语言中一个字符串的输入是一个前缀），是“前缀隔离（prefix isolated）”（输入的前缀是语言中的一个字符串）的对偶。如果我们正在找一个前缀，那我们一般会想找到最长的可能的前缀。这可以通过标记最近的被识别为前缀的位置*P*，然后继续解析直到我们到达结尾或者出现错误被卡住。那么*P*就是最长前缀的末尾。

许多定向解析器使用前瞻方式，这意味着即便在输入的末尾，也必须有足够的令牌用于前瞻。这可以通过引入一个输入结尾令牌来实现，例如**#**或其他任何语法中没有的令牌。对于一个使用*k*个令牌做前瞻的解析器，*k*个**#**的副本将会被追加到输入字符串中；解析器的前瞻机制将会也会进行相应修改；例子参见9.6节。唯一的接受状态就是第一个**#**即将被接受的状态，而这通常也表示解析完成了。

这大大简化了目前的情形和上面的表格，因为现在解析器在输入没有结束时不能处于可接受状态。这去掉了上表中前缀的两个答案。然后我们将上表的上下部分进行叠加，然后最左边的列就变得多余了。就变成了下面这张表：

can	in an accepting state?	
continue?	yes	no
yes	—	continue
no	OK (yes)	error in input / premature EOI (no)

然后我们将区分“错误输入”和“过早结束”的工作交给错误报告机制来完成。

由于在定向解析器中没有明确的终止标准，所以每个解析器都有自己的停止标准，这是一个有点不理想的状态。这本书中，我们将使用最终输入标记，只要它有助于终止，并且当然，对于解析器使用前瞻机制。

3.9 传递闭包

解析中的许多算法（以及在计算机科学中的许多其他分支）都具有一些从初始信息开始的属性，然后根据一些理论规则推到得出结论，一直到不能得出更多的结论为止。在2.9.5.1 和2.9.5.2 节中的推理规则中, 我们已经看到了两个例子。这些推理是完全不同的，并且一般的推理规则可以是任意复杂的。为了得到一个清晰的计算结论的算法，闭包算法，我们现在将考虑一个最简单的可能的推理规则: 传递性。这种规则的形式如下：

如果*A*□*B* 并且*B*□*C* 那么*A*□*C*

其中□是任意符合规则的操作符。最明显的是=，但是<和□以及其他许多也是可以的。但是□（不等于）却不是的。

作为一个例子，我们将考虑一个非终结符的“左角集”的计算。一个非终结符*B*在一个非终结符*A*的左角集中，如果有一个派生*A*□*B*···，知道这点有时是有用的，因为在其他方面来说，任何*B*可以开始的字符*A*也都能开始。

有下面这样一个语法

```
Sₛ  →  S T
S   →  A a
T   →  A t
A   →  B b
B   →  C c
C   →  x
```

我们如何才能找出**C**在**S**的左角集中？语法中的规则**S**□**ST**和**S**□**Aa**立即就让我们知道**S**和**A**在**S**的左角集中。我们把这写作**S∠S**和**A∠S**，其中∠代表左角。这同样告诉我们**A∠T**,**B∠A**，还有**C∠B**。这是我们的初始信息（图3.19（a））。

(a)	(b)	(c)	(d)
S∠S	S∠S ✔	B∠S ✔	C∠S ✔
A∠S	A∠S ✔	C∠S	
A∠T	A∠S ✔	C∠T	
B∠A	B∠S	C∠S ✔	
C∠B	B∠T	C∠T ✔	
	B∠S ✔		
	B∠T ✔		
	C∠A		
	C∠A ✔		

Fig. 3.19. The results of the naive transitive closure algorithm

现在很容易看出，如果**A**在**B**的左角集中，**B**在**C**的左角集中，那么**A**也在**C**的左角集中。公式如下：

A∠B ∧ B∠C □ A∠C

这是我们的推理规则，而且我们将使用它来得出新结论或“推论”，通过两两组合已知的事实来产生更多的已知因子。然后通过应用推理规则直到不再产生新的因子来获得传递闭包。在传递闭包的上下文中，因子也被称为“关系”，虽然一般∠是（二进制）关系，并且A∠B和B∠C 是关系的 "实例"。

通过图 3.19 (a) 中的列表，我们首先将**S∠S**和**S∠S**结合起来。这将产生**S∠S**,这是相当令人失望的因为我们已经知道了；它在图 3.19 (b) 中, 标有一个√, 以表明它不是新的。（**S∠S, A∠S**的）结合产生**A∠S**，但是我们也已经知道了。没有其他的因子与**S∠S**结合，所以我们继续看**A∠S**，而这得到了**A∠S**和**B∠S**；第一个我们已经知道了，但是第二个是第一次被我们知道。接着（**A∠T**,**B∠A**）就得到了**B∠T**，等等，做完剩下部分的第一轮结果见图3.19(b)。

第二轮结合了三个有新有旧的因子。第一个是发现由**A∠S**和**C∠A (c)**得到了**C∠S**，第二个发现是**C∠T**。

第三轮将（c）中的两个新因子与（a），（b），（c）中的结合起来，但没有发现新的因子；所以这个算法最终发现了10个因子。

请注意，我们已经在这个初级算法中实现了一次优化：基础算法将启动第二轮甚至更多轮次，通过将已知的所有因子之间配对，而不仅仅只是在新发现的因子之间。

在一个图中用弧线表示因子或关系通常是有帮助的。最初的情况见图3.20(a)，最终的结果见（b）。箭头旁边的数字表明得到该因子经过了几轮计算。

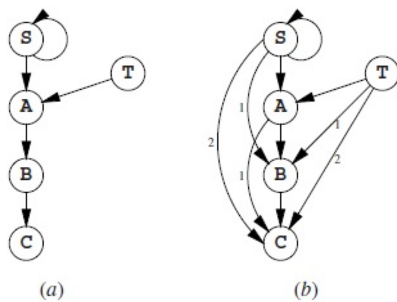


Fig. 3.20. The left-corner relation as a graph

闭包算法的效率很大程度上依赖于它所所使用的推理算法，而传递规则的情况被广泛的进行了研究。传递闭包主要有三种方法来进行：初级版、普通版和高级版；我们将简要的对每一种进行介绍。上面描述过的初级算法，在通常情况下往往是相当有效率的，但会画出一张很大的图，而且特殊情况下可能会需要计算很多轮。同样它还会重复计算很多次，我们可以在图3.19中看出；15个结果中有10个是已经得到了的。但考虑到“正常”语法的大小，初级算法几乎可以满足所有情况下的解析。

普通的进行传递闭包的方式是使用Warshall的算法[409]。其优势是非常简单实现，而且它需要的时间仅取决于图中N节点的数量而不是弧线的数量，但它的缺点是总是需要O（N_3）的时间。这回让它在和其他任何闭包算法的比较中总是输掉。

高级算法避免了导致初级算法效率底下的劣势：1.图中的圆圈被收缩为“强联通分量”；2.弧线以一种顺序组合起来，并允许弧线进行复制而不是重复计算；3.使用更有效的数据表示。例如，一个高级算法首先会计算从**A**的所

有输出弧，然后将之拷贝至**T**而不是重新计算一次。Tarjan [334] 描述了第一个高级可传递闭包算法。并在其他很多刊物上广泛转载；见Nuutila[412]和互联网。它需要的时间与其最终得出的结果数量成正比。

高级可传递闭包算法在大型应用程序 (数据库等) 中非常有用，但它在解析中的位置确实令人怀疑的。一些作者建议在LALR解析器生成器中使用它们，但应该在非常庞大的语法上使用，以保证算法的复杂性有一个很好的回报。

强调算法的闭包性质的优点可以让人集中于推理规则，并将底层的闭包算法当做理所当然；这对于算法设计很有帮助。然而大多数的解析算法都很简单，以至于不需要分解成推理规则和闭包解释。因此我们将仅在有助于理解的地方使用推理规则（9.7.1.3节），以及当其原本就是语法的一部分的情况下使用（7.3节，图表解析）。对于其余的我们将简单的讲述算法，然后指出他们是传递闭包算法得出的。

3.10 解析与布尔矩阵乘法的关系

在解析和布尔矩阵乘法之间有一个显著但又有点神秘的关系，因为很可能把一个转换为另一个，带有很多可能和但是。这有很有趣的含义。

一个布尔矩阵是一个其中所有元素只能为0或者1的矩阵。比如如果一个矩阵T的索引代表镇，那么元素$T_{i,j}$可能就表示城镇i到城镇j的直达铁路的距离。这样的矩阵可以和另外一个矩阵$U_{j,k}$相乘，这可以表示比如，一个城镇j到城镇k的直达巴士的距离。而$V_{i,k}$（T和U的乘积）的结果是一个布尔矩阵，这代表从城镇i到城镇k是否能联通，首先考虑火车然后才是巴士。这立刻就可以展示$V_{i,k}$是怎样计算出来的：必须有一个1，如果存在一个j使得$T_{i,j}$和$U_{j,k}$都能有保有一个1，否则就必须有一个0。公式如下：

$$V_{i,k} = (T_{i,1} \wedge U_{1,k}) \vee (T_{i,2} \wedge U_{2,k}) \vee \cdots \vee (T_{i,n} \wedge U_{n,k})$$

其中∧是布尔运算的和，∨是布尔运算的或，n是矩阵的大小。这意味着$O(n)$的行为被V的每次输出所要求，其中就有n^2；所以这个算法的时间复杂度为$O(n^3)$。

图3.21给了一个示例；矩阵$T_{2,}$ </sub>的行和矩阵$U_{*,2}$的列结合得到了矩阵$V_{2,2}$的输出。布尔矩阵的乘法运算中，是不遵守交换律的：可以大致的理解为从一个镇到另一个镇有一个火车-巴士的路线，但是没有巴士-火车的路线，也就是T×U不等于U×T*。注意这个跟传递闭包是不同的：在传递闭包中，一个单一关系遵循无限的次数，而在布尔矩阵运算中则是一个关系结果之后才有第二个。

$$\overset{T}{\begin{vmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{vmatrix}} \times \overset{U}{\begin{vmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{vmatrix}} = \overset{V}{\begin{vmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{vmatrix}}$$

Fig. 3.21. Boolean matrix multiplication

上述是布尔矩阵乘法（*BMM*）的一个小实战，实际上BMM在许多数学和工科分支中是非常重要的，并且关于如何高效的使用它有一个完整的学科[1]。数十年的集中努力带来了一系列越来越高效和负责的算法。V. Strassen[2]是第一个打破$O(n^3)$阻碍的算法，使用的是$O(n^{2.81\cdots})$算法，到了现在这个记录是$O(n^{2.376\cdots})$；时间是1987年。很明显，至少$O(n^2)$次运行是必须的，但目前看来这个时间复杂度是达不到的。

对我们的论点来说更重要的是，在1975年Valiant [18]展示了如何将一个CP解析问题转换为一个BMM问题。特别的是，如果可以在$O(n^k)$次操作中将两个$n \times n$的布尔矩阵相乘，那就可以在$O(n^k)+O(n^2)$步中解析一个长度为n的字符串，其中$O(n^2)$是转换的成本。因此，我们可以在$O(n^{2.376\cdots})$中进行一般的CF解析，这确实比CYK算法的立方时间依赖性要好。但是Valiant算法和快速BMM算法都太过复杂和耗时，所以这种方法只有当输入字符串以百万计甚至更多的时候才拿来使用比较好。更要命的是，它要求所有的输入都必须存储在内存中，因为它是无方向的算法，因此它使用的数据结构的大小是$O(n^2)$，这意味着它只能在内存单位的TB的计算机上运行。简而言之, 它的意义只是理论上的。

在2002年Lee [39]显示了一个BMM问题如何转换为一个CF解析问题。特别的是，如果你可以在$O(n^{3-\delta})$次操作中对一个长度为n的字符串进行常规CF解析，那就能在$O(n^{3-\delta/3})$次操作中将两个大小为$n \times n$的布尔矩阵相乘。那就再次出现了$O(n^2)$的转换成本，但由于δ最多可为2（不太可能在$O(n)$中解析一个实例），$O(n^{3-\delta/3})$的步骤至少是$O(n^2)$，这决定了$O(n^2)$；要注意δ = 0的情况$O(n^3)$通常是多个问题的边界。Lee的转换所涉及的计算工作量比那些用Valiant算法的要小的多，所以一个真正快速的通用CF解析算法可能会提供一个快速实用的BMM算法。这样一个快速通用的CF算法必须是非BMM并且比$O(n^3)$具有更高的时间复杂性；但不幸的是目前还不知道这样的算法。

一般的CF和布尔矩阵乘法有共同之处，其最佳算法的效率是未知的。图Fig 3.22总结了各种可能性。x轴表示最佳CF算法的效率；y轴表示最佳BMM算法的效率。图中的位置表示了这些值的组合。由于这些值是未知的，我们不知道图中的哪个点对应于实际，但是我们可以排除几个区域。

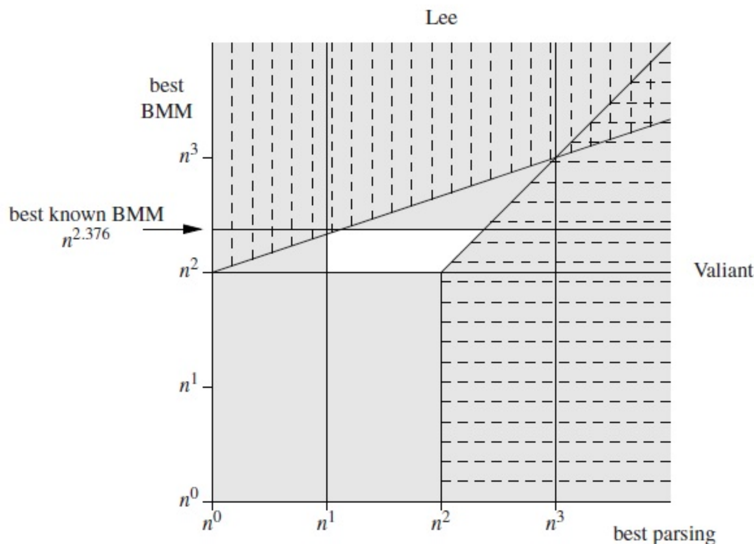


Fig. 3.22. Map of the best parser versus best BMM terrain

在现有算法的基础上灰色区域被排除。例如，在n^3的垂直线的右侧灰色部分由于CYK算法被排除了，它在$O(n^3)$中进行一般的CF解析；所以其组合（最好的解析器，最好的BMM）不能有一个大于$O(n^3)$的第一组件。同样的，n^1垂直线的左侧区域代表了小于$O(n)$工作量的算法，而这是不可能的因为解析器必须到达每个令牌。BMM要求至少$O(n^2)$次动作，但已经有$O(n^{2.376\cdots})$这个算法可以用了；这产生了两个水平禁区。

阴影标记部分是被Valiant和Lee转换算法排除的部分。Valiant的结果不包括右侧的水平阴影部分；Lee的结果不包含顶部的垂直阴影部分。真正的最佳解析和BMM算法的组合只能在中间的白色无阴影区域中。

对BMM算法的广泛研究并没有产生一个比$O(n^3)$更实用的算法；由于BMM可以被转换为解析，因此可以解释为什么关于CF一般算法的没那么广泛的研究没有产生比$O(n^3)$更实用的算法，除了通过BMM。另一方面，图Fig 3.22显示了普通CF解析还是有可能成为线性的（$O(n^1)$），以及BMM可能比$O(n^2)$更糟糕。

Rytter [34]已经将普通CF解析和一个特定形式在一个格子中用最短路径连接起来，通过某种含义。

Greibach [389]描述了“最难的上下文无关语言”，是一种语言，如果我们能在时间$O(n^x)$内解析它，那我们就在$O(n^x)$时间内解析任何语言。不用说，肯定很难解析。本文隐式的使用了一种很少被注意到的解析技术；见问题3.7。

> [1]. For a survey see V. Strassen, “Algebraic complexity theory”, in Handbook of Theoretical Computer Science, vol. A, Jan van Leeuwen, Ed. Elsevier Science Publishers, Amsterdam, The Netherlands, pp. 633-672, 1990. ↩

> [2]. V. Strassen, “Gaussian elimination is not optimal”, Numerische Mathematik, 13:354-356, 1969. ↩

3.11 总结

语法允许通过一个明确的过程产生句子，而这个过程的细节决定了句子的结构。解析通过模拟产生过程（自顶向下解析）或回滚（自底向上解析）来恢复这个结构。眞正的工作是收集信息来指导有效的恢复结构的过程。

有一个完全不同的--令人惊讶的--无语法方式来进行CF解析，“面向数据分析”，这在本书的范围之外。见Bod [348]以及互联网。

问题

问题**3.1**：假设给定语法中的所有终结符是不同的。语法是否明确？

问题**3.2**：编写一个程序, 给定一个语法G和一个数字n, 计算G允许的不同的有n个叶子(终端)的解析树的数量。

问题**3.3**：如果您熟悉现有分析器 (生成器), 请标识其分析器组件, 如69页所述。

问题**3.4**：3.5.4 节中的迷宫预处理算法消除了所有具有三壁的房间；在确定性迷宫中有两面或四面墙的是可以接受的。那么没有墙或者只有一面墙的房间呢？它们如何影响算法和结果？消除掉它们是否可能/有用？

问题**3.5**：构造一个例子，使得一个确定性自底向上解析器，对于某个*k*，在位置*k*必须执行*k*次操作。

问题**3.6**：项目：图 3.10*(b)*中的迷宫有几种可能的路径，因此一个迷宫定义了一组路径。很容易就可以看出这些路径形成了一个常规集合。这样一个迷宫等同于一个语法。深入挖掘一下这个类比，例如：1.从迷宫的某些描述中推到出语法规则。2.子集算法（5.3.1节）如何变化迷宫？3.是否有可能生成一组迷宫，它们一起可以定义一个给定的CG集合？

问题**3.7**：项目: 研究 Greibach [389] 的 "平移和交叉匹配括号" 解析方法。

问题**3.8**：展示图3.14的一个版本，其中在顶部附近标记为2的节点联合了在输入中不支持的解析数。

问题**3.9**：实现3.7.3 中草绘的回溯算法的蓝图。

问题**3.10**：假设算法表达式用一个高度模糊的语法解析，其中对数字进行了适当的定义。设计一个条件可以帮助优化解析林，以获取服从运算符一般优先权的解析树。例如，4+5×6+8应该是((4+(5×6))+8)这样的。考虑到前四个运算符是左结合，但幂运算↑是右结合运算符：6/6/6可以写成((6/6)/6)，但6↑6↑6却是(6↑(6↑6))。

```
Expr_S   → Number
  Expr   → Expr Operator Expr | ( Expr )
Operator → + | - | × | / | ↑
```

问题**3.11**：研究项目：一些解析问题包含了非常大的CF语法，有数百万的规则。这样的语法是由程序生成的，并且将会把有限的上下文条件合并到语法中。它通常是非常多余的，包含许多相似的规则，并且非常不精确。许多常规CF解析器在语法范围内是二次的，上千万的规则带来10^{14}的因子。能找到一个解析技术可以在这样的语法上很好的工作吗？（另请参见问题4.5。）

问题**3.12**：扩展项目：

1. 对于一个令牌对(t_1,t_2)如果S中有$\#t_1=\#t_2$并且S的前缀都有$\#t_1 \geq \#t_2$，那么一个字符串S是平衡的，其中$\#t$是S或其前缀中t出现的次数。一个令牌对(t_1,t_2)对语法G来说是括号对，如果所有$L(G)$的字符串对于(t_1,t_2)来说都是平衡的。设计一种算法来检查令牌对(t_1,t_2)是否是括号对，对于给定语法G：a)在简化但合理的假设下，括号对一起出现在规则的右侧（例如**F** □ **(E)**），b)普通情况下。

2. 在字符串中位置i的令牌t_1和位置j的令牌t_2相匹配，如果它们之间的字符串段$i+1 \cdots j-1$对于(t_1,t_2)是平衡的。一个括号对(t_1,t_2)和另一个括号对(u_1,u_2)是兼容的，如果$L(G)$字符串中每一个段对于一个t_1以及其匹配的t_2，对于(u_1,u_2)都是平衡的。证明如果(t_1,t_2)与(u_1,u_2)兼容，那么(u_1,u_2)与(t_1,t_2)兼容。

3. 对于一个给定语法，设计一个算法来找到最大的兼容括号对集合。

4. 使用括号对集合来构造$L(G)$中的句子，在线性时间内。

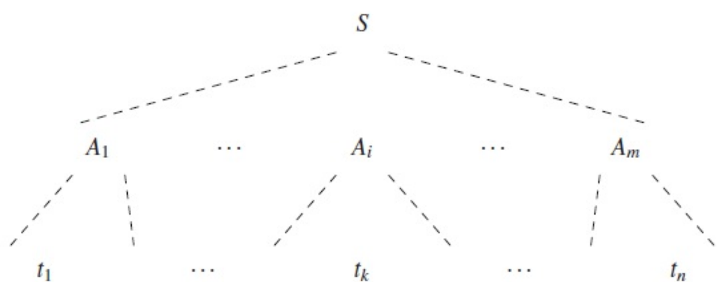
5. 从G中获取有关$L(G)$中不以这种形式构造的字符串段的信息，例如正则表达式。

6. 设计更进一步的技术来利用CF语言的括号对蓝图。

4 一般非定向分析

在这章中我们将会介绍两种解析方法，都是无向的：Unger法和CYK法。这些方法被称为无向性，因为它们以看似任意的方向接受输入。它们要求在解析开始之前，所有的输入都存储在内存中。

Unger方法是自顶向下的；如果输入属于这个语言，则必须从语法的起始符号开始衍生，比如S。因此，它必须从起始符号的右侧开始衍生，比如$A_1A_2...A_m$。这反过来又意味着A_1必须可以推倒出输入的第一部分，A_2必须可以推倒出第二部分，等等。如果输入的句子是$t_1t_2...t_n$，这个需求可以描述如下：



Unger方法试图找到适合这个需求的输入的分区。这是一个递归问题：如果一个非终结符A_i要推导出输入的某个部分，则这部分的一个分区必须适应A_i的右侧。最终，这样的右侧必须由仅有终结符号组成，并且这些可以很容易与当前的输入部分相匹配。

CYK方法用另一种方法来解决这个问题：它试图在输入的右侧中找到出现的部分；每当找到一个，它就在推导出这一部分的左侧的位置标记一下。用相对应的左侧来替换右侧出现的部分，结果会产生输入的一些句子形式。这些句子形式再次成为查询右侧的对象。最终，我们可能会找到一个句子形式，可以同时派生输入句子和属于起始符号的右侧。

在接下来的两节中，将会对这些方法进行详细介绍。

4.1 Unger解析方法

Unger解析方法[12]由一个CF语法和一个输入句子组成。我们将首先讨论Unger解析方法的语法部分，不含ε规则和循环（见2.9.4节）。然后在讨论引入ε规则之后所带来的问题，并对解析方法进行修改以适应所有的CF语法。

4.1.1 不含ε规则和循环的Unger解析方法

为了了解Unger方法如何解决解析问题，让我们举一个小例子。假设我们有一个语法规则：

$S \rightarrow ABC \mid DE \mid F$

并且我们想知道*S*是否推导出输入句子*pqrs*。然后初始解析问题可以用如下示意图来表现：

S
pqrs

对于每个手册，我们必须首先生成输入的所有可能的分区。生成分区并不难：如果我们有*m*个杯子，编号从1到*m*，有*n*个大理石，编号从1到*n*，我们必须找到所有的分区，使得一个杯子至少装有一个大理石，每个杯子中大理石的编号都是连续的，并且小编号杯子所含大理石编号比大编号杯子所含大理石编号要小。我们这样来做：首先我们将1号大理石放在1号杯子里，然后将其余的*n*-1个大理石和*m*-1个杯子全部分区。这就让我们有了全部的分区，在第一个杯子中有且只有大理石1的情况。接下来，我们把大理石1、2放在第一个杯子中，然后在对剩下的*n*-2个大理石和*m*-1个杯子进行分区，如此继续下去。如果*n*小于*m*，则不存在分区。

划分输入相当于用杯子（右侧的标志）来划分大理石（输入符号）。如果一个右侧有比句子更多的符号，那就找不到任何分区（没有ε规则）。对于右侧的第一个标志，那分区则必须像以下这样：

S		
A	*B*	*C*
p	*q*	*rs*
p	*qr*	*s*
pq	*r*	*s*

第一个子分区产生了以下子问题：*A*是否派生出*p*，*B*是否派生出*q*，*C*是否派生出*rs*？这些问题的答案都必须是肯定的，否则分区就是错误的了。

对于第二个右侧，我们得到以下分区：

S	
D	*E*
p	*qrs*
pq	*rs*
pqr	*s*

对于最后一个右侧，可以得到以下分区：

S
F
pqrs

所有这些子问题都涉及到较短的句子，除了最后一个。它们都将导致类似的拆分，到最后许多都会失败因为右侧的终结符无法与对应部分的分区相匹配。唯一会引起关注的分区是最后一个。它和我们开始的那个一样复杂。这就是我们不允许语法中存在循环的原因。如果语法中存在循环，我们可能就得一次又一次的重复原来的问题。例如，如果上面的示例中存在一个*F*→*S*的规则，那一定会出现这种情况。

以上说明我们这里有一个搜索的问题，我们可以用深度优先搜索或者广度优先搜索技术（见3.5.2节）来解决它。Unger方法使用深度优先搜索来解决。

在接下来的讨论中，图4.1的语法将作为一个例子。这个语法代表了简单的算数表达式语言，包括运算符**+**和**×**，以及运算数**i**。

```
Expr_S  ->  Expr + Term | Term
  Term  ->  Term × Factor | Factor
Factor  ->  ( Expr ) | i
```

Fig. 4.1. A grammar describing simple arithmetic expressions

我们将使用句子**(i+i)×i**作为输入示例。因此最初的问题就表现为：

Expr
(i+i)×i

将**Expr**的第一个替代项写入**(i+i)×i**的输入，得到一个15个分区的表，见图Fig4.2。在这里我们不对全部进行讨论，虽然算法的优化版本需要这样做。Fitting the first alternative of Expr with the input (i+i)×i results in a list of 15 partitions, shown in Figure 4.2. We will not examine them all here, although the unoptimized version of the algorithm requires this.We will only examine the partitions that have at least some chance of succeeding: we can eliminate all partitions that do not match the terminal symbol of the right-hand side. So the only partition worth investigating further is:

Expr		
Expr	+	Term
(	i	+i)×i
(	i+	i)×i
(	i+i	)×i
(	i+i)	×i
(	i+i)×	i
(i	+	i)×i
(i	+i	)×i
(i	+i)	×i
(i	+i)×	i
(i+	i	)×i
(i+	i)	×i
(i+	i)×	i
(i+i	)	×i
(i+i	)×	i
(i+i)	×	i

Fig. 4.2. All partitions for Expr→Expr+Term

Expr		
Expr	+	Term
(i	+	i)×i

The first sub-problem here is to find out whether and, if so, how Expr derives (i. We cannot partition (i into three non-empty parts because it only consists of 2 symbols. Therefore, the only rule that we can apply is the rule Expr--->Term. Similarly, the only rule that we can apply next is the rule Term--->Factor. So we now have

Expr
Term
Factor
(i

However, this is impossible, because the first right-hand side of Factor has too many symbols, and the second one consists of one terminal symbol only. Therefore, the partition we started with does not fit, and it must be rejected. The other partitions were already rejected, so we can conclude that the rule Expr--->Expr+Term does not derive the input.

The second right-hand side of Expr consists of only one symbol, so we only have one partition here, consisting of one part. Partitioning this part for the first right-hand side of Term again results in 15 possibilities, of which again only one has a chance of succeeding:

Expr		
Term		
Term	×	Factor
(i+i)	×	i

Continuing our search, we will find the following derivation (the only one to be found):

```
Expr →
Term →
Term × Factor →
Factor × Factor →
( Expr ) × Factor →
( Expr + Term ) × Factor →
( Term + Term ) × Factor →
( Factor + Term ) × Factor →
( i + Term ) × Factor →
( i + Factor ) × Factor →
( i + i ) × Factor →
( i + i ) × i
```

This example demonstrates several aspects of the method: even small examples require a considerable amount of work, but even some simple checks can result in huge savings. For example, matching the terminal symbols in a right-hand side with the partition at hand often leads to the rejection of the partition without investigating it any further. Unger [12] presents several more of these checks. For example, one can compute the minimum length of strings of terminal symbols derivable from each non-terminal. Once it is known that a certain non-terminal only derives terminal strings of length at least n, all partitions that fit this non-terminal with a substring of length less than n can be immediately rejected.

5 正则语法与有限状态

正则语法，又称为3类语法，是最简单的语法形式，仍具有生成力。他们可以描述串联（连接两个字符串在一起）和重复，并可以指定替代方案，但他们却无法表达嵌套。正则文法可能是正式的语言学的最好理解的部分，并且 可以回答关于他们的几乎所有问题。

5.1 正则语法的应用

除了他们的简洁性之外，正则语法还有许多的应用，其中我们将简要地提及最重要的部分。

5.1.1 CF解析中的正则语言

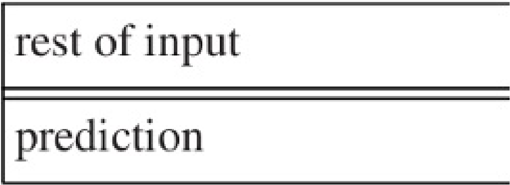
在某些 CF 语法分析器中，子分析器能够被分辨出是处理正则文法的 Such a subparser is based implicitly or explicitly on the following surprising phenomenon. Consider the sentential forms in leftmost or rightmost derivations. Such sentential forms consist of a closed (finished) part, which contains terminal symbols only and an open (unfinished) part which contains non-terminals as well. In leftmost derivations the open part starts at the leftmost non-terminal and extends to the right; in rightmost derivations the open part starts at the rightmost nonterminal and extends to the left. See Figure 5.1 which uses sample sentential forms from Section 2.4.3.



It can easily be shown that these open parts of the sentential form, which play an important role in some CF parsing methods, can be described by a regular grammar, and that that grammar follows from the CF grammar.

6 一般的有向自顶向下解析 (Directional Top-Down Parsing)

在这一章中，我们将要讨论用预测(prediction)尝试重新推导出输入句子的自顶向下解析方法。正如3.2.1中解释的那样，我们从开始符号开始，尝试从它产生输入句子；在每一时刻，我们有一个句型代表我们对剩余输入句子的预测。将预测画在它所预测的输入句子的那部分正下面会很方便，左端对齐，就像我们在图3.5中做的那样：



这种句型由终结符和非终结符组成。如果是一个终结符在最前面，我们将它跟当前的输入字符匹配。如果是一个非终结符在最前面，我们选择它的右侧中的一个，用那个右侧替换该非终结符。这样，我们总是替换掉最左的非终结符，最后，如果成功了的话，我们已经模仿了最左推导。注意到预测部分跟做最左推导时句型的开放部分(open part)是对应的，就像5.1.1讨论的那样。

6.1 模仿最左推导

现在让我们用一个例子解释这样一个推导过程。考虑图6.1所示语法。这个语法生成所有有相同数量a和b的句子。

$$
\begin{aligned}
S_S &\rightarrow aB \mid bA \\
A &\rightarrow a \mid aS \mid bAA \\
B &\rightarrow b \mid bS \mid aBB
\end{aligned}
$$

Fig. 6.1. A grammar producing all sentences with equal numbers of **a**s and **b**s

让我们从开始符号S开始尝试解析句子aabb。S是我们的第一个预测.我们的预测的第一个符号是非终结符，所以我们要用它的一个右侧替代它。在这个语法中，S有两个选择：我们或者用规则S→aB，或者用规则S→bA.句子从a开始而不是b，所以这里我们不能用第二个规则。应用第一个规则我们得到预测aB.现在预测的第一个字符是一个终结符。这里，我们没有选择：

a	abb
a	B

我们要将这个字符和句子的当前字符做匹配，这里当前字符也是a。所以匹配，接受a。这留给我们预测B和剩余的句子：abb。预测的第一个字符又是一个非终结符，所以要被它的一个右侧代替。现在我们有三种选择。然而，第一和第二个选择在这里不适用，因为它们由b开始，但我们需要a。所以，我们采用第三个选择，现在我们有了预测aBB：

a	a	bb
a	a	BB

我们再一次匹配了当前的输入字符，所以我们接受它，然后继续从预测BB去匹配bb。我们要再次将最左的B用它的右侧替代。句子的下一个字符是b，所以这里第三个选择不适用。这仍然留给我们两个选择：b和bS.所以我们可以两个都尝试，或者用一点更聪明的方法。如果我们采用bS,我们会得到至少一个a（因为S），所以这不可能是正确的选择。因此我们选择b，于是得到了对应bb的预测bB。我们再次得到了一个匹配结果，这留给我们预测B和句子b。由于同样的原因，我们选择b。匹配之后，我们的预测为空。幸运的是，我们同时到达了输入句子的末尾，所以我们接受这个句子。如果我们记录下了使用过的产生式，就能得到下面的推导过程：

S→aB→aaBB→aabB→aabb

图6.2用树的形式展现了解析的步骤。图中的虚线将已经处理过的部分和未处理部分分开。每个图中，预测部分最左边的符号被处理。 这个例子显示了本章讨论的解析器的几个共同点：

- 我们总是对预测中的最左符号进行处理
- 如果这个符号是终结符，我们没有选择：它和当前的输入字符匹配或者解析失败

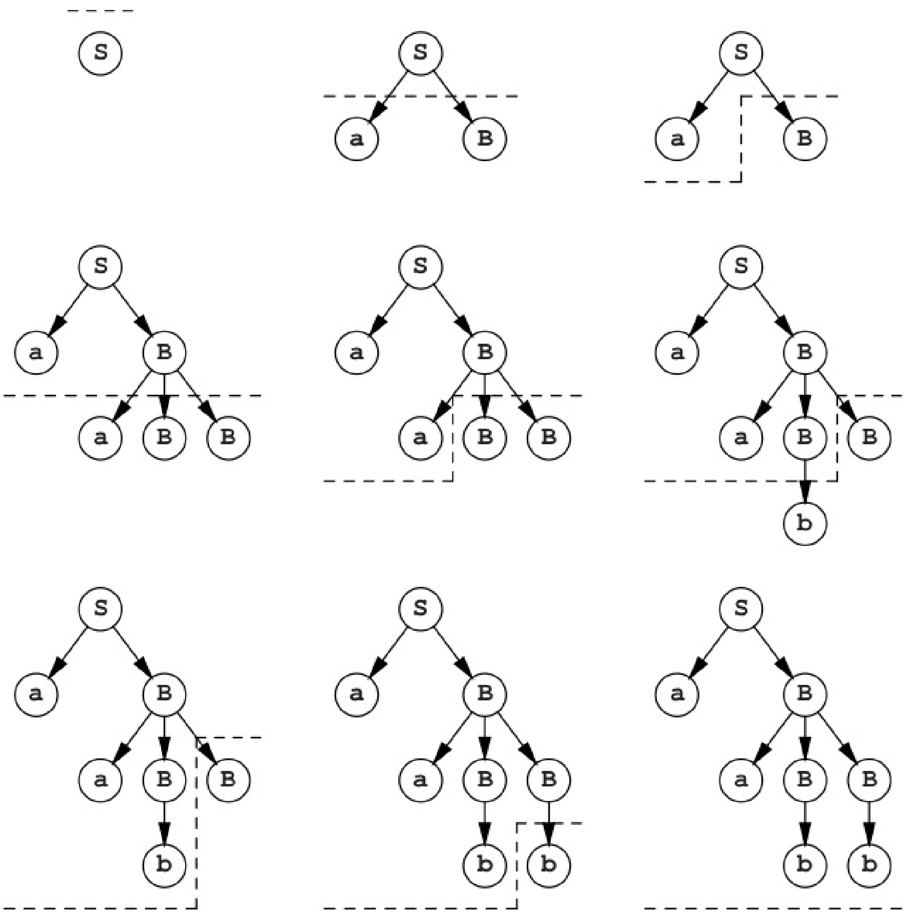


Fig. 6.2. Production trees for the sentence **aabb**

- 如果这个符号是非终结符，我们需要做出一个猜测：它需要被它的一个右侧替代。因此，我们总是先处理预测中的最左符号，从而得到了最左推导。
- 所以，自顶向下方法将解析树的节点用前序组织：父节点在它的儿子之前被识别。

6.2 下推自动机(pushdown automaton)

在上面的例子中我们进行的步骤跟所谓的下推自动机很相似。下推自动机（PDA）是一个假想的数学装置，它读入输入串，用栈（stack）进行控制。栈内可以包含属于栈符号表（stack alphabet）的符号。栈是一种只能从一端被使用的列表：最后一个进入（“pushed”）列表的是第一个被取出（“popped”）的。栈有时被叫做先进先出表（first-in,last-out list 或 FILO list）：第一个进去的符号最后一个出来。在上面的例子中，预测的工作方式就像一个栈，这也就是下推自动机使用栈的目的。因此我们将这个栈称作预测栈（prediction stack）。同时，这个栈解释了术语“下推”自动机：为了后续过程，自动机在栈上“推入”符号。

下推自动机的工作方式是弹出一个栈符号然后读一个输入符号。接着这两个符号一般会提供我们几种将被推入的栈符号串的一个选择。所以存在一个（输入符号，栈符号）二元组到栈符号串列表的映射。在栈空并且达到输入符号尾的时候，自动机接受输入的句子。如果选择存在多种（所以（输入符号，栈符号）映射到不止一个的串上），当存在一些选择使得读到句尾时栈空时，自动机接受句子。

这种自动机是以满足格雷巴赫范式（Gretbach Normal Form, GNF）的上下文无关语法为模型的。在这种范式中，所有语法规则都满足A→a或$A \rightarrow aB_1B_2...B_n$,这里a是一个终结符，$A,B_1,...,B_n$是非终结符。栈符号当然是终结符。$A \rightarrow aB_1B_2...B_n$形式的规则对应着（a,A）对$B_1B_2...B_n$的映射。这意味着如果输入符号是a，预测栈栈顶是A，我们可以接受a，将预测栈上的A替换为串$B_1B_2...B_n$。A→a对应着（a,A）向空串的映射。开始时，这个自动机栈顶为语法的开始符号。每一个不产生空串的上下文无关语法都能转化为格雷巴赫范式（Greibach[8]）。大多数形式语言理论的书都讨论了如何做这项工作（例如见Hopcroft and Ullman[391]）。

图6.1的示例语法已经是格雷巴赫范式了，所以我们轻松的从它构建一个下推自动机。这个自动机在图6.3用映射的形式表示了出来。

```
(a, S) → B
(b, S) → A
(a, A) →
(a, A) → S
(b, A) → AA
(b, B) →
(b, B) → S
(a, B) → BB
```

Fig. 6.3. Mapping of the PDA for the grammar of Figure 6.1

这里有很重要的一点需要注意：很多下推自动机是非确定（non-deterministic）的。比如，图6.3所示的下推自动机可以为（a,A）选择空串或S。事实上，存在不能构建出确定的下推自动机的上下文无关语言，尽管我们能构建出非确定的。

还需要提及的是：这里我们讨论的下推自动机是自动机理论里的的简化版。在自动机理论里，下推自动机有状态（state），相应的映射是从（状态，输入符号，栈符号）三元组到（状态，栈符号列表）的二元组。从这个角度看，它们像是用一个栈拓展了的有限状态自动机（第五章讨论的）。而且，下推自动机有两种类型：一些用栈空表示接受句子，另一些用在标注为接受状态（accepting state）的状态处结束表示接受。也许这点会令人惊讶，拥有状态并不能让下推自动机这个概念更有力（powerful），有状态的下推自动机仍然只能接受可以被上下文无关语法描述的语言。在我们的讨论中，下推自动机只有一个状态，所以我们已经忽略它了。

如果我们想将其转化为解析用的自动机（parsing automata），以上描述的下推自动机有几个缺点需要克服。首先，下推自动机需要我们将我们的语法转化为格雷巴赫范式。虽然语法转换对形式语言学家不是问题，我们希望避免它们，尽量用原始的语法。现在我们稍微放宽格雷巴赫范式的要求，允许栈符号是终结符，然后对所有a，在映射关系中添加 (a,a)→ 这样我们就能使用所有右侧都由终结符开始的任何语法了。同时，我们可以将下推自动机的工作步骤分为“匹配”和“预测”，就像我们在6.1节中的例子做

的那样。“匹配”步骤对应(a,a)→的使用，“预测”步骤对应(,A)→...，这个就是说，栈上的非终结符被它的一个右侧替代，不用消耗输入符号。对图6.1所示语法，这样表时候的结果如图6.4所示，这事实上仅仅就是重写了图6.1的语法。

```
 (, S) → aB
 (, S) → bA
 (, A) → a
 (, A) → aS
 (, A) → bAA
 (, B) → b
 (, B) → bS
 (, B) → aBB
(a, a) →
(b, b) →
```

Fig. 6.4. Match and predict mappings of the PDA for the grammar of Figure 6.1

我们之后将会看到，即使用了这种方法，我们还是有可能必须改动语法，但现在这看起来非常前途有望，所以我们采用这种策略。这个策略同时解决了另一个问题：ε规则不再需要特殊的处理了。为了得到格雷巴赫范式，我们要除去它们但是这不再需要了，因为它们刚好对应(,A)→

第二个缺点是这个下推自动机没有保存使用过的规则（映射）的记录。因此，我们在这个自动机中引入一个解析栈（analysis stack）。对每一个预测的步骤，我们将被替换的非终结符，以使用的右侧的序号（将右侧从1到n标上序号）为后缀推入解析栈。这样，解析栈刚好跟图6.2中虚线左边的部分对应，这个虚线表示了解析栈与预测栈的分隔。这导致了这个自动机在每个时候都有如图6.5所示结构。这个结构，跟当前状态，栈等等，被称作一个即时描述（instantaneous description）。在图6.5中，匹配过程可以本看作将竖直线向右推。

matched input	rest of input
analysis	prediction

Fig. 6.5. An instantaneous description

然而最重要的第三个缺点，是非确定性。形式上，这个自动机当且仅当有一个选择的序列使得在句尾的时候栈空接受句子，这个说法可能让人满意，但是对于我们的目的不是这样，因为这没有告诉我们怎么得到这个序列。我们必须引导自动机去做正确的选择。回看6.1节的例子，在推导的很多时候需要做出选择，而且我们的选择基于了一些对当前的语法的特殊的考虑：有时我们关注句子的下一个符号，也有时我们看得更远，去确定之后没有a出现。在这个例子里，选择是容易的，因为所有右侧由终结符开始。然而一般的，找到正确的选择要困难的多。比如右侧可以由一个非终结符开始，它又有从非终结符开始的右侧，等等。

在第八章我们将看到，给定句子的下一个符号，很多语法仍然允许我们决定使用哪一个右侧。但是在本章中，我们将关注能在更大一部分的语法上工作的自顶向下解析方法。而不是尝试基于特殊考虑来决定选择，我们要从所有可能出发来引导自动机。在第三章我们曾看到，在解决在特定的点有多种分支的问题的时候一般有两种方法：深度优先搜索和广度优先搜索。我们将看到如何完成这两种方法的机器操作。由于操作的步数可能会按输入规模以指数级别增长，即使是一个小例子也可能有庞大的作业量。为了让事情更有趣，我们将只用图3.4所示的固有歧义的语言，这个语法在图6.6这里重复了一遍。我们将用aabc作为测试输入。

$$
\begin{array}{lcl}
S_S & \rightarrow & AB \mid DC \\
A & \rightarrow & a \mid aA \\
B & \rightarrow & bc \mid bBc \\
D & \rightarrow & ab \mid aDb \\
C & \rightarrow & c \mid cC
\end{array}
$$

Fig. 6.6. A more complicated example grammar

6.3 广度优先自顶向下解析（Breadth-First Top-Down Parsing)

用于自顶向下解析问题的广度优先方法是去维护一个所有预测的列表。然后每一个预测都像上面6.2节说的那样处理，也就是说，如果栈顶是一个非终结符，预测栈就被一些新的预测栈替换，新的预测栈数量跟它所对应的选择数相同。在这每一个新栈里，栈顶的非终结符被对应的选择替换。预测步骤对所有预测栈施用（包括新生成的），直到所有预测栈栈顶都是终结符。这时对所有预测栈，我们将最前面的终结符与当前输入符号进行匹配，将不匹配的预测栈删去。如果没有预测栈剩下来，这个句子不属于该语言，所以，我们的自动机现在维护一组（stack,analysis stack）而不是一个，其中每一个对应一个可能选择，如图6.7所示。

matched input	rest of input
analysis1	prediction1
analysis2	prediction2
...	...

Fig. 6.7. An instantaneous description of our extended automaton

这个方法对在线解析（on-line parsing）同样适用，因为它按从左到右的顺序处理输入串。从左到右处理输入并且得到最左推导（leftmost derivation）的解析方法被称为LL解析方法。第一个L代表从左到右，第二个代表最左推导。依此我们基本知道了该怎么写出这样一个解析器，但现在我们还有一个细节没有恰当地处理：终止。当最终我们得到了一个空的预测栈时，输入句子就属于该语法描述的语言吗？仅当输入被读完才是！为了避免这个额外的检查，并且避免出现已经达到句尾但解析没有停止的

情况，我们引入一个特殊的终止记号（*end marker*）**#**.这个终止记号附加在句子和预测的末尾，所以当它们成功匹配时我们就能知道这个预测匹配了输入句子，解析成功。

6.3.1 一个例子

(*a*)

	aabc#
	S#

(*b*)

	aabc#
S_1	DC#
S_2	AB#

(*c*)

	aabc#
S_1D_1	abC#
S_1D_2	aDbC#
S_2A_1	aB#
S_2A_2	aAB#

(*d*)

a	abc#
S_1D_1a	bC#
S_1D_2a	DbC#
S_2A_1a	B#
S_2A_2a	AB#

(*e*)

a	abc#
S_1D_1a	bC#
$S_1D_2aD_1$	abbC#
$S_1D_2aD_2$	aDbbC#
$S_2A_1aB_1$	bc#
$S_2A_1aB_2$	bBc#
$S_2A_2aA_1$	aB#
$S_2A_2aA_2$	aAB#

(*f*)

aa	bc#
$S_1D_2aD_1a$	bbC#
$S_1D_2aD_2a$	DbbC#
$S_2A_2aA_1a$	B#
$S_2A_2aA_2a$	AB#

(*g*)

aa	bc#
$S_1D_2aD_1a$	bbC#
$S_1D_2aD_2aD_1$	abbbC#
$S_1D_2aD_2aD_2$	aDbbbC#
$S_2A_2aA_1aB_1$	bc#
$S_2A_2aA_1aB_2$	bBc#
$S_2A_2aA_2aA_1$	aB#
$S_2A_2aA_2aA_2$	aAB#

(*h*)

aab	c#
$S_1D_2aD_1ab$	bC#
$S_2A_2aA_1aB_1b$	c#
$S_2A_2aA_1aB_2b$	Bc#

(*i*)

aab	c#
$S_1D_2aD_1ab$	bC#
$S_2A_2aA_1aB_1b$	c#
$S_2A_2aA_1aB_2bB_1$	bcc#
$S_2A_2aA_1aB_2bB_1$	bBcc#

(*j*)

aabc	#
$S_2A_2aA_1aB_1bc$	#

Fig. 6.8. The breadth-first parsing of the sentence aabc

图6.8展示了对句子aabc的一个完整的广度优先解析过程。最开始只有一个预测栈：它包含开始符号和终止记号;没有符号被接受（a）。得到（b）的步骤是一个预测步骤;有两个可能的右侧，所以我们得到两个预测栈。预测栈的不同也表现在了分析栈上，S的不同后缀代表不同的右侧选择。下一个有多种右侧的预测步骤得到了（c）。现在所有预测栈栈顶都是终结符;刚好都成功匹配，得到（d）。接下来，有些预测开头又是非终结符了，所以进行下一个预测步骤得到（e）。下一个步骤是匹配步骤，幸运的是，有些匹配失败了;这些会被去掉，因为由它们不可能得到成功的解析。从（f）到（g）又是一个预测步骤。另一个有一些失败匹配的匹配步骤让我们得到（h）。更远的一个预测步骤导致（i）然后接着一个匹配步骤最终给我们带来（j），终止记号匹配，解析成功。

解析（analysis）是

$S_2A_2aA_1aB_1bc\#$

目前，我们不需要解析里的终结符;把它们去掉得到

$S_2A_2A_1B_1$

这说明我们通过先应用规则S_2，再A_2等等得到最左推导，每次替代掉最左的非终结符。检查一下：

S→AB→aAB→aaB→aabc

这里描述的广度优先最初被Greibach [7]陈述。但是，在那个陈述中，语法被转化为了格雷巴赫范式，并且使用的步骤跟我们最开始的下推自动机的相似。预测和匹配步骤被结合起来了。

6.3.2 一个反例：左递归（Left recursion）

上述方法明显适用于那个语法，那么问题是它是否对所有上下文无关语法都适用呢？有的人可能认为是，因为在任何出现的预测里，对所有非终结符，所有可能都被有系统地尝试了。不幸的是，这个推理有个严重的缺点，这被下一个例子说明了：让我们看看句子ab是否属于这个简单语法定义的语言： S→Sb | a 我们的自动机从下面的状态开始：

	ab#
	S#

因为在预测的开始是一个非终结符，我们进行预测步骤，得到：

	ab#
S_1	Sb#
S_2	a#

有一个预测再次以非终结符开始，我们再进行预测：

	ab#
S_1S_1	Sbb#
S_1S_2	ab#
S_2	a#

现在，很清楚发生了什么事：我们好像最终进行了一个无限的过程，什么都得不到。导致这样的原因是我们一直在尝试S→Sb规则，从未达到可以尝试匹配的状态。无论何时，有一个非终结符能推导出无限多的由非终结符开始的句型，这个问题就会发生，进而没有匹配步骤会发生。因为这无

限多的句型由非终结符开始，非终结符的数量又是有限的，就至少有一个非终结符在开头出现过超过一次。所以我们有：A→…→Aα.一个能推导出由自己开始的句型的非终结符就叫做左递归。

左递归有几种类型：当有一个A→Aα这样的语法规则，我们称为直接左递归（*immediate left recursion*），就像规则S→Sb;当递归通过了其他规则，比如A→Bα,B→Aβ，我们称为间接递归（*indirect left recursion*）。这些形式的左递归都可能被ε-产生式隐藏;这分别导致隐藏左递归（*hidden left recursion*）和隐藏间接左递归（*hidden indirect left recursion*）。比如在这个语法里

```
S → ABc
B → Cd
B → ABf
C → Se
A → ε
```

非终结符S,B,和都是左递归的。有左递归的非终结符的语法也称为左递归语法。

一个语法没有ε规则也没有循环（loop）的情况下，如果我们使用一个额外的步骤，我们仍然可以使用我们的解析策略：如果一个预测栈有超过输入句子长度数量的符号，它不可能推导出输入句子（每一个非终结符推导出至少一个符号），所以它能被去掉。然而，这个小技巧有一个大弊端：它需要提前知道输入句子的长度，所以这个方法不再适用于在线解析。幸运的是，左递归能被消除：给定一个左递归语法，我们可以将它转化为一个定义相同语言，并且没有左递归非终结符的语法。鉴于对任何自顶向下解析方法，左递归都是一个主要问题，我们将要讨论这个语法转化。

6.4 消除左递归

我们先讨论消除直接左递归的方法。假定ε-规则和单元规则（unit rule）已经被消除了（见4.2.3.1和4.2.3.2）。现在，令A是一个左递归的非终结符（原文为rule），并且

$$A \rightarrow A\alpha_1 \mid \cdots \mid A\alpha_n \mid \beta_1 \mid \cdots \mid \beta_m$$

是A的所有规则。没有等于ε的α_i，否则我们会有A→A，一个单元规则。也没有等于ε的β_j，否则会有一个ε-规则。A只用$A \rightarrow A\alpha_k$规则生成的句型有这样的形式：

$$A\alpha_{k_1}\alpha_{k_2}\cdots\alpha_{k_j}$$

并且当$A \rightarrow \beta_i$规则使用时，句型不再以A开头，对一些i，和一些$k_1,\ldots,k_j$,它有如下的形式：

$$\beta_i\alpha_{k_1}\alpha_{k_2}\cdots\alpha_{k_j}$$

这里j可能为0.同样的句型可以被如下规则生成：

$$\begin{aligned} A_{head} &\rightarrow \beta_1 \mid \cdots \mid \beta_m \\ A_{tail} &\rightarrow \alpha_1 \mid \cdots \mid \alpha_n \\ A_{tails} &\rightarrow A_{tail}\, A_{tails} \mid \varepsilon \\ A &\rightarrow A_{head}\, A_{tails} \end{aligned}$$

或者，使得没有新的ε规则重新生成的话：

$$\begin{aligned} A_{head} &\rightarrow \beta_1 \mid \cdots \mid \beta_m \\ A_{tail} &\rightarrow \alpha_1 \mid \cdots \mid \alpha_n \\ A_{tails} &\rightarrow A_{tail}\, A_{tails} \mid A_{tail} \\ A &\rightarrow A_{head}\, A_{tails} \mid A_{head} \end{aligned}$$

这里A_{head},A_{tail}和A_{tails}是新引入的非终结符。没有α_i是ε，所以A_{tail}不会推导出ε，所以A_{tails}不是左递归。A可能仍然是左递归的，但不是直接左递归，因为没有β_j以A开始。然而，它们可能推导出以A开始的句型。 一般的，消除间接左递归要更复杂。思路就是先将非终结符标号，标为$A_1,A_2,...,A_n$.现在，对一个左递归非终结符A，有一个推导

$$A \rightarrow B\alpha \rightarrow \cdots \rightarrow C\gamma \rightarrow A\delta$$

，每时每刻句型的最左边都是非终结符，然后再三地用它的一个右侧替代。每一个非终结符都有一个标号，写作$i_1,i_2,...,i_m$,于是在推导中我们得到了这么一串数：$i_1,i_2,...,i_m,i_1$.现在，如果我们没有任何$A_i \rightarrow A_j\alpha (j \le i)$,这就是不可能的，因为$i_1<i_2<...<i_m<i_1$是不可能的。 现在就要消除这样的规则。我们从$A_1$开始。对$A_1$，要消除了只是直接递归的规则，我们已经看到了应该怎么做。接着轮到A_2。每一个有着$A_2 \rightarrow A_1\alpha$形式的产生式都要被替代成

$$A_2 \rightarrow \alpha_1\alpha \mid \cdots \mid \alpha_m\alpha$$

这里A_1的规则为

$$A_1 \rightarrow \alpha_1 \mid \cdots \mid \alpha_m$$

这不可能产生$A_2 \rightarrow A_1\gamma$形式的新规则，因为我们已经消除了$A_1$的左递归规则，而且$\alpha_i$都不为ε。接着，我们消除$A_2$的直接递归规则。这样对$A_2$的工作就结束了。类似的，我们对$A_3$到$A_n$进行处理，按照总是先替代$A_i \rightarrow A_1\gamma$,再$A_i \rightarrow A_2\delta$等等的顺序。我们必须按照这样的顺序，因为，替换一个$A_i \rightarrow A_2\delta$的规则可能会引入$A_i \rightarrow A_3\gamma$这样的规则，而不会是$A_i \rightarrow A_1\alpha$

6.5 深度优先（回溯）解析器

上一节叙述的广度优先方法有一个缺点，它会占用大量内存。深度优先方法也有一个不足，它的一般形式不适用于在线解析。然而，有很多应用场景下不需要在线解析，于是深度优先方法就比较有利，因为它不需要大量内存。

在深度优先方法中，当我们面临多种可能选择时，选择一个，暂时搁置其他可能。首先，充分地检查我们的选择产生的结果。如果选择被证明是错误的（或者是成功的，但我想要全部的解），就回滚我们的操作直到现在的状态，然后用其他的可能选择继续。

让我们看看这个搜索技术是怎么应用于自顶向下解析的。我们的深度优先解析器遵循跟广度优先解析器一样的操作，直到遇到一个选择：预测栈顶的非终结符有超过一个右侧。现在，选择第一个右侧，而不是建立一个新的解析栈/预测栈。这个选择通过在涉及到的非终结符附上下标1,来反映到解析栈上，跟我们在广度优先解析器那做的一样。然而这次，解析栈不仅仅用作记录解析过程，还用作回溯。

解析器继续这个过程，直到发生匹配失败，或者终止记号成功匹配。如果预测栈是空的，我们就找到了一个解析，它被解析栈的内容所描述;如果匹配失败，解析器会回溯。这个回溯包含如下步骤：首先，解析栈顶的所有终结符出栈，并依次推入预测栈。并且，这些符号从已匹配输入中移出，并加在剩余输入的开始处。这是“匹配”步骤的逆操作。所以遍及终结符的回溯通过向后移动竖线完成，就像图6.9那样。

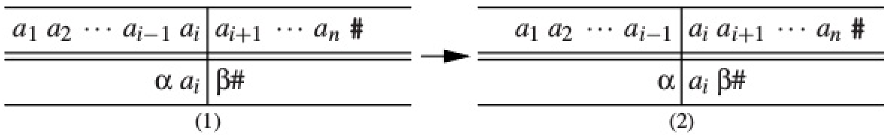


Fig. 6.9. Backtracking over a terminal

那么有两种可能：如果解析栈空，就没有其他可能可以尝试，解析结束;否则，解析栈顶是一个非终结符，并且预测栈顶对应了它的一个右侧。选择

这个右侧导致了失败的匹配。这种情况下，我们从解析栈弹出这个非终结符，用它替换预测栈顶对应它的右侧的部分，就像图6.10那样。

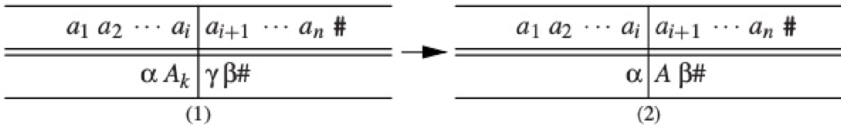


Fig. 6.10. Backtracking over the choice for the k-th rule for A, $A \to \gamma$

接下来有两种可能：如果那是这个非终结符最后一个右侧，我们已经尝试过它的全部右侧，需要进一步回溯;如果不是，我们再次开始解析，首先用它的下一个右侧替换掉这个非终结符。

现在让我们尝试解析句子aabc，这次使用回溯解析器。图6.11一步步展示了这个过程;回溯步骤用B标注了出来。这个例子表露了回溯方法的另一个缺陷：它可能会做出错误的选择，在很久之后才会发现。

就像这里展示的，解析过程会在解析被找到的时候停止。如果我们想找到全部的解析，在终止记号匹配时，我们不应该停止。我们可以使用回溯来继续，就好像没有找到成功的解析，并且在每次匹配终止记号时记录下解析栈（它代表了解析过程）。最后，我们的解析栈变空，表示我们已经穷尽了所有可能，这时解析停止。

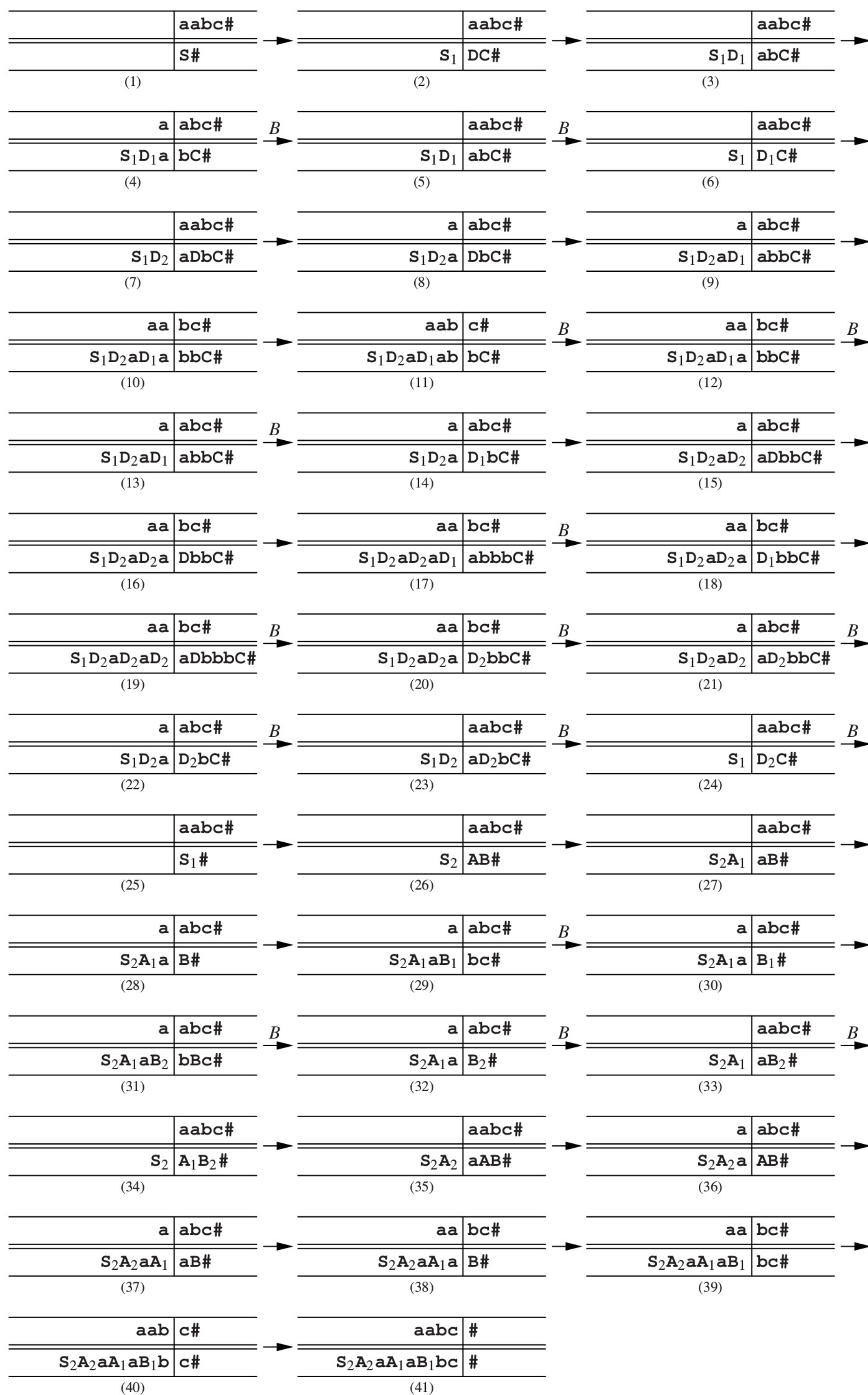


Fig. 6.11. Parsing the sentence **aabc**

6.6 递归下降

在前面的小节里，我们已经见过了几种自动机的工作，在处理输入句子时用语法决定解析的步骤。现在有另一种方式的说明：这些自动机将语法作为程序。将一个语法看作解析机器的一个程序并不像看起来那么牵强。毕竟，一个语法是推导其所描述的语言的句子的指示，我们在自顶向下解析中做的就是从一个语法重新推导一个句子。这跟将语法看作生成装置的传统观点的差别仅在于我们现在在尝试重推导一个特定的句子，而不是任意句子。以这种方式看，语法就是程序，以一种描述性的编程语言写成——就是说，这种语言定义结果而不是获得结果需要进行的步骤。

如果想写一个给定的上下文无关语法的解析器，我们有两种选择。第一种是按照前几节里描述的自动机是写一个程序。这个程序可以以一个语法和一个句子为输入。这听起来很完美，而且易于实现。困难来自于当解析器必须在部分输入被识别时进行一些其他操作的时候。比如，当处理了一个声明语句，编译器须要生成符号表。对效率的考量与这一点一起引导出第二个选项：对给定语法写一个专用的解析器。有很多这样的专用解析器，其中的大多数都用了一个叫做递归下降的识别技术。我们将假设读者具有一定的编程经验，明白过程和递归。如果不是，这一节可以跳过。本节没有描述新的解析方法，只描述了一个实现的技术，它常用于手写解析器和一些机器生成的解析器。